RAISING THE BAR:
ADVANCING MITIGATIONS AGAINST MEMORY-CORRUPTION AND SIDE-CHANNEL ATTACKS

TOMMASO FRASSETTO

Cumulative Dissertation
submitted in fulfilment of the requirements
for the degree of Doktor-Ingenieur (Dr.-Ing.)

First Ph.D. Referee:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi

Second Ph.D. Referee:
Prof. Mauro Conti

TECHNISCHE
UNIVERSITÄT
DARMSTADT

System Security Lab
Fachbereich Informatik
Technische Universität Darmstadt

Darmstadt 2022

Tommaso Frassetto:
Raising The Bar: Advancing Mitigations Against Memory-Corruption and Side-Channel Attacks

Darmstadt, Technische Universität Darmstadt

Year thesis published in TUprints	2022
URN of the Dissertation	`urn:nbn:de:tuda-tuprints-214363`
Date of the Defense	May 19, 2022

ABSTRACT

The complexity of computer programs has been increasing for multiple decades. As a result, the number and impact of security vulnerabilities have been rising as well. Memory-corruption attacks have been one of the most severe security threats for decades, despite the tremendous efforts of the security community: according to data published by Microsoft in 2019, 70% of vulnerabilities addressed in security updates during the previous decade are memory safety issues. Similarly, according to the 2021 CWE Top 25, two out of the three most dangerous vulnerability categories are related to memory corruption.

A variety of approaches have been proposed that aim to either discover vulnerabilities before they are deployed to a production environment, or to mitigate vulnerabilities by making them harder to exploit. The former case includes strategies like static analysis, test suites, and fuzzing. While these methods are important and beneficial, it is not feasible to find *all* software bugs: most deployed software projects suffer from memory-corruption vulnerabilities, in particular if they contain legacy code.

Hence, it is crucial to investigate, develop, and deploy *mitigations*, in order to make exploitation of these vulnerabilities substantially harder or even infeasible. Three prominent approaches are *software diversity* (e.g., randomization), *integrity checks* (e.g., CFI), and *memory isolation* (e.g., TEEs). The scope of this cumulative dissertation includes contributions to these three mitigation approaches, as well as applications to more practical problems.

The idea of software diversity is to change the protected program so that one or more of its properties, e.g., the address of code or data, is unknown to the attacker. Since memory-corruption exploits depend on the address of code and data, the adversary needs to correctly guess or otherwise acquire the address of all the memory structures required for the exploit, which substantially increases the attack's difficulty. We present *Selfrando*, a scheme which randomizes the executable code of a program with a fine granularity. In contrast to previous works, our scheme performs the randomization every time the application is launched. This way, the same application package can be distributed through traditional channels to all users, but each run has a different memory layout. *Selfrando* was successfully integrated in the privacy-preserving Tor Browser and deployed in the hardened version of Tor Browser for Linux.

The principle of *Control-Flow Integrity* (CFI) is to instrument indirect control flow transfers to inspect the computed target before the control transfer is performed. As an example of a coarse-grained CFI scheme, indirect call instructions can be instrumented to only allow calls to the start addresses of known functions. A variety of CFI approaches with different granularity have been proposed. Given this variety, it is important to be able to quantify the security guarantees of each scheme. We present *CFInsight*, a benchmark for CFI security. Unlike previous works, our analysis is based on properties of the paths between indirect call sites and system call instructions, which attackers need to reach in order to interact with the operating system or the file system. Our metric is based on quantitative measurements of these paths and indicates how hard it is to construct an attack.

Memory isolation involves introducing barriers between various software components, so that a vulnerability in one of them cannot be used to exploit a different one. Memory isolation can be implemented purely in software or with the help of hardware extensions, e.g., *Trusted Execution Environ-*

ments (TEEs). Memory isolation is particularly beneficial for software that is composed by a significant number of diverse components, especially if some of them handle untrusted data. This is the case of web browsers, where an attacker can target the *Just-In-Time* (JIT) compiler and force it to generate malicious code. We present *JITGuard*, which leverages memory isolation and a TEE in order to protect the internal data of a browser's JIT compiler from attackers. Unlike alternative approaches, our construction maintains the existing synchronous calling semantics and does not require complex redesigns in the original code.

Memory isolation, and TEEs in particular, can be used for a number of purposes, including to ensure the confidentiality of a component. However, there is a category of attacks that is particularly effective in breaking the confidentiality property of memory isolation. These attacks leverage *side channels* in order to extract information from a component without directly interacting with it. Below, we introduce our proposals of a software-only and a hardware-based countermeasure against these attacks.

A traditional approach to limit side-channel leakage is to manually design alternative algorithms, which requires significant expertise and is highly error-prone. Instead, we present *DR.SGX*, a software-only solution that automatically protects all data used in a TEE by applying a fine-grained location randomization. The data location is periodically re-randomized to further limit the leakage during extended execution.

Another common approach to limit cache-based side-channel leakage is to partition the caches, which leads to high performance overheads. Our proposal *HybCache* is a new cache architecture that limits side-channel leakage by design. Security-sensitive code accesses a subset of the cache fully-associatively, using a random replacement policy, which prevents any address-specific information leakage. At the same time, security-insensitive code accesses the cache in the traditional set-associative way, which, unlike cache partitioning, results in no performance degradation.

Lastly, we describe two practical use cases that show how TEEs can be used to improve protocols. First, we show *VoiceGuard*, a TEE-based design for a voice recognition system that protects from disclosure both the user's voice data and the vendor's machine learning model. Second, we present *FastKitten*, a TEE-based protocol that allows for fast and efficient smart contract execution on cryptocurrencies that do not support smart contracts.

ZUSAMMENFASSUNG

Die Komplexität von Computerprogrammen nimmt seit Jahrzehnten zu. Infolgedessen haben auch die Anzahl und die Auswirkungen von Sicherheitslücken zugenommen. Memory-Corruption Angriffe sind seit Jahrzehnten eine der schwerwiegendsten Sicherheitsbedrohungen, trotz der enormen Anstrengungen der Security-Community: Nach 2019 veröffentlichten Statistiken von Microsoft sind 70% der in den letzten 10 Jahren behobenen Schwachstellen in Sicherheitsupdates Probleme mit der Speicherintegrität. Auch in den CWE Top 25 von 2021 stehen zwei der drei gefährlichsten Schwachstellenkategorien im Zusammenhang mit Speicherkorruption.

Es wurde eine Vielzahl von Ansätzen vorgeschlagen, die darauf abzielen, entweder Schwachstellen zu entdecken, bevor sie in einer Produktionsumgebung eingesetzt werden, oder Schwachstellen zu verhindern, indem man sie schwerer ausnutzbar macht. Zum ersten Fall gehören Strategien wie statische Analyse, Testsuiten und Fuzzing. Solche Methoden sind zwar wichtig und vorteilhaft, allerdings ist es nicht realistisch alle Softwarefehler zu finden: Die meisten Softwareprojekte leiden unter Memory-Corruption Schwachstellen, insbesondere wenn sie Legacy-Code enthalten.

Daher ist es entscheidend, Schutzmaßnahmen zu erforschen, zu entwickeln und einzusetzen, um das Ausnutzen solcher Schwachstellen erheblich zu erschweren oder sogar unmöglich zu machen. Drei prominente Ansätze dazu sind Software-Diversität (z.B. Randomisierung), Integritätsprüfungen (bspw. CFI) und Speicherisolierung (z.B. TEEs). Im Rahmen dieser kumulativen Dissertation werden Beiträge zu diesen drei Maßnahmen, aber auch Anwendungen für praktischere Probleme, präsentiert.

Die Idee von Software-Diversität besteht darin, das zu schützende Programm so zu verändern, dass eine oder mehrere seiner Eigenschaften, z.B. die Adresse von Code oder Daten, dem Angreifer nicht zugänglich sind. Da Speicherkorruptionen von der Adresse des Codes oder der Daten abhängen, muss der Angreifer alle Speicheradressen, welche für den Angriff erforderlich sind, entweder erraten oder auf andere Weise in Erfahrung bringen. Dies erhöht die Schwierigkeit des Angriffs erheblich. Mit Selfrando stellen wir einen Ansatz vor, welcher den ausführbaren Code eines Programms feingranular randomisiert. Im Gegensatz zu früheren Veröffentlichungen, führt unser Ansatz die Randomisierung jedes Mal durch, wenn die Software gestartet wird. Auf diese Weise kann das gleiche Programmpaket über herkömmliche Kanäle an alle Benutzer verteilt werden, aber jede Ausführung bekommt ein anderes Speicherlayout. Selfrando wurde erfolgreich in den privaten Tor-Browser integriert und mit der gehärteten Version des Tor-Browsers für Linux ausgerollt.

Das Prinzip von Control-Flow Integrity (CFI) besteht darin, indirekte Kontrollflusstransfers zu instrumentieren, um das berechnete Ziel zu prüfen bevor der Kontrollflusstransfer durchgeführt wird. Als Beispiel für einen groben CFI-Ansatz können indirekte Call-Instruktionen instrumentiert werden, um nur Aufrufe an die Startadressen bekannter Funktionen zu erlauben. Es wurde eine Vielzahl von CFI-Ansätzen mit unterschiedlicher Granularität vorgeschlagen. Angesichts dieser Vielfalt ist es wichtig, die Sicherheitsgarantien der einzelnen Verfahren quantifizieren zu können. Dazu stellen wir CFInsight vor, ein Benchmark für CFI-Sicherheit. Im Gegensatz zu vorherigen Ansätzen basiert unsere Analyse auf den Eigenschaften der Pfade zwischen indirekten Call-Instruktionen und System-Calls, die Angreifer erreichen müssen, um mit dem Betriebssystem oder dem Dateisystem zu interagieren. Unsere Metrik

basiert auf quantitativen Messungen dieser Pfade und gibt an, wie schwer es ist, einen Angriff zu konstruieren.

Bei der Speicherisolierung werden Barrieren zwischen verschiedenen Softwarekomponenten eingesetzt, so dass eine Schwachstelle in einer dieser Komponenten nicht zur Ausnutzung einer anderen genutzt werden kann. Die Speicherisolierung kann ausschließlich in Software, oder mit Hilfe von Hardware-Erweiterungen, z.B. Trusted Execution Environments (TEEs), realisiert werden. Speicherisolierung ist besonders vorteilhaft für Software, die aus einer beträchtlichen Anzahl verschiedener Komponenten besteht, insbesondere wenn einige dieser Komponenten ungeprüfte Daten verarbeiten. Dies ist bei Webbrowsern der Fall, bei denen ein Angreifer den Just-In-Time-Compiler (JIT) angreift und ihn zwingen kann, bösartigen Code zu erzeugen. Daher stellen wir JITGuard vor, welches Speicherisolierung und ein TEE nutzt, um die internen Daten des JIT-Compilers eines Browsers vor Angreifern zu schützen. Im Gegensatz zu alternativen Ansätzen behält unsere Konstruktion die bestehende synchrone Aufrufsemantik bei und erfordert keine komplexen Abänderungen im ursprünglichen Code.

Speicherisolierung und insbesondere TEEs können für eine Reihe von weiteren Zwecken eingesetzt werden, unter anderem um die Vertraulichkeit einer Komponente zu gewährleisten. Es gibt jedoch eine Kategorie von Angriffen, mit denen die Vertraulichkeit der Speicherisolierung besonders effektiv gebrochen werden kann. Diese Angriffe nutzen Seitenkanäle aus, um Informationen aus einer Komponente zu extrahieren, ohne direkt mit dieser zu interagieren. Im Folgenden stellen wir unsere Vorschläge für rein software- und hardwarebasierte Gegenmaßnahmen gegen diese Angriffe vor.

Ein traditioneller Ansatz zur Einschränkung von Seitenkanal-Lecks besteht in der manuellen Entwicklung alternativer Algorithmen, was erhebliche Fachkenntnisse erfordert und höchst fehleranfällig ist. Stattdessen stellen wir DR.SGX vor, eine reine Softwarelösung, die automatisch alle in einem TEE verwendeten Daten durch eine feingranulare Randomisierung des Speicherlayouts schützt. Das Speicherlayout der Daten wird in regelmäßigen Abständen neu randomisiert, um das Datenleck während längerer Laufzeiten weiter zu begrenzen.

Ein anderer gängiger Ansatz zur Begrenzung von Cache-basierten Seitenkanal-Lecks ist die Partitionierung der Caches, was zu einer reduzierten Leistung führt. Bei unserem Projekt HybCache handelt es sich um eine neue Cache-Architektur, welche gezielt Seitenkanal-Lecks begrenzt. Dabei greift der sicherheitskritische Code auf eine Teilmenge des Cache vollständig assoziativ zu, wobei eine zufällige Replacement-Policy verwendet wird, die ein adressenspezifisches Informationsleck verhindert. Gleichzeitig greift unkritischer Code auf traditionelle Weise auf den Cache zu, was im Gegensatz zur Cache-Partitionierung keine Leistungseinbußen zur Folge hat.

Abschließend beschreiben wir zwei praktische Anwendungsfälle, die zeigen, wie TEEs zur Verbesserung von Protokollen eingesetzt werden können. Zunächst präsentieren wir VoiceGuard, ein TEE-basiertes Design für ein Spracherkennungssystem, das sowohl die Sprachdaten des Benutzers als auch das maschinelle Lernmodell des Anbieters vor der Offenlegung schützt. Zweitens stellen wir FastKitten vor, ein TEE-basiertes Protokoll, das eine schnelle und effiziente Ausführung von Smart Contracts in Kryptowährungen ermöglicht, welche keine Smart Contracts unterstützen.

CONTENTS

Part I

SYNOPSIS

1 INTRODUCTION

In recent decades, computer systems evolved from rare and specialized equipment used by technicians to omnipresent devices deployed in every branch of human activities — including for industrial, financial and military purposes. The complexity of both the hardware used in these systems and the programs that run on them increased dramatically, leading to a similar growth in vulnerabilities[1]. At the same time, adversaries have an increasing number of attack classes at their disposal to help them achieve their goals.

One of the most effective ways to exploit a system is to leverage so-called *memory-corruption vulnerabilities*[2], which allow the attacker to access the memory of a running program in an unintended way. Despite the tremendous amount of effort by the research community, this type of vulnerability is still very dangerous[3].

Memory-corruption vulnerabilities are especially common in languages like C and C++ because these languages require the programmer to manually manage memory allocations, which leads to the possibility of programming mistakes. As an example, the most straightforward example of a memory-corruption attack is to leverage a *buffer overflow*: when copying data from a buffer to another, a missing or faulty check on the length of the data being copied allows the attacker to overwrite memory locations after the end of the buffer. This can change the value of variables in memory and even change the control flow by overwriting a code pointer. More advanced attacks also leverage other vulnerabilities, e.g., *use after free*. In Figure 1 we show a number of software and hardware components that are relevant to the attacks and mitigations we consider. As an example, Application A has a vulnerable buffer ❶ that can be used to attack a particular target ❷.

Researchers have proposed a number of approaches to mitigate memory-corruption vulnerabilities. The ideal solution would be to find all the vulnerabilities and patch them, which would clearly eradicate the problem. Bug-finding strategies, including static analysis, test suites, and fuzzing, can and should be used to find and correct software vulnerabilities. However, finding *all* bugs is not feasible in practice due to the complexity of modern software. Hence, it is crucial to investigate, develop, and deploy *mitigations*, in order to make exploitation of these vulnerabilities substantially harder or even unfeasible. Three prominent approaches are *software diversity* (e.g., randomization), *integrity checks* (e.g., CFI), and *memory isolation* (e.g., TEEs). Each of these techniques has advantages and shortcomings regarding their effectiveness and efficiency.

In this dissertation we consider a number of attacks against a complex program, and propose and evaluate mitigation strategies against them based on these three approaches. We will now briefly explain each of these techniques and summarize our contributions to the state of the art.

1 As an example, in the year 2000 just over 1 000 vulnerabilities were discovered and publicly disclosed. In 2021 the same figure grew to over 20 000 [149].

2 In 2019, Microsoft showed that 70% of the vulnerabilities addressed by their security updates over the previous decade are memory-safety issues [138].

3 According to the *2021 CWE Top 25* [147] — a list of the vulnerability classes connected with the highest number of the CVE vulnerabilities over the previous two years — three of the top seven vulnerabilities are instances of memory-corruption attacks. Namely, number 1 is *Out-of-bounds Write*, number 3 is *Out-of-bounds Read*, and number 7 is *Use After Free*.

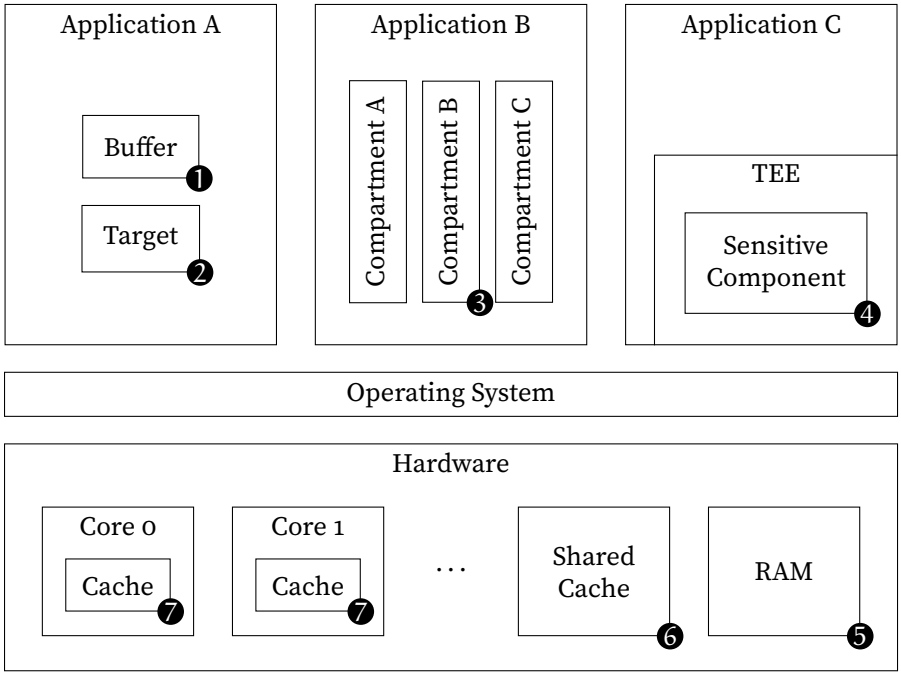

Figure 1: Complex applications running on a modern processor.

1.1 SOFTWARE DIVERSITY

A *software diversity* defense approach consists in producing multiple versions of the same program. All versions share the same exact logic, but their implementations are different, and the adversary does not know the specific version the victim uses. Assuming that each version requires a custom exploit, i.e., that no generic attack exists, the exploit only succeeds if the adversary picks the correct attack for the specific version, which is very unlikely if a sufficiently large number of versions exist. A concrete way to implement this idea is to *randomize* the memory layout of the program, i.e., change the order of the different blocks of code that compose the program. In the case of Application A in Figure 1, the use of randomization would mean that the adversary would not know where the target 2 is. This would make the attack unfeasible, unless the adversary is able to disclose the location of the target.

Although randomization is a well-known technique, previous approaches suffer from a number of downsides: either they are very coarse-grained, and hence, easier to bypass, or they are performed at compile time, such that a user's version of the program does not change with time.

SELFRANDO Our contribution *Selfrando* [1, Appendix A] lifts both of those restrictions, delivering a fine-grained randomization scheme that is performed on the user's device every time the protected application is started. As a result, Selfrando provides substantially higher entropy than ASLR, while remaining compatible with conventional software build and distribution pipelines (unlike compile-time randomization approaches [65]). As a result of this practicality, we successfully integrated Selfrando with Tor Browser, a privacy-preserving Web browser. Selfrando was also included in the hardened version of Tor Browser for Linux [67].

1.2 CONTROL-FLOW INTEGRITY

Another approach to mitigate memory-corruption vulnerabilities is to make sure each code fragment can only call a function if the programmer intended the fragment to be able to call that function. This is called *Control-Flow Integrity* (CFI) [24]. As a result, the adversary is unable to change the control flow and force the application to perform malicious actions. If CFI is deployed in Application A in Figure 1, overwriting a code pointer would still be possible, but the malicious pointer would be rejected at run time and execution would be interrupted.

It is important to note that CFI by design does not detect data-only attacks, which only change a program's data without introducing detectable deviations to the control flow. The two other approaches we consider, randomization (Section 1.1) and memory isolation (Section 1.3) can be effective against data-only attacks. An extension of CFI, *Data-Flow Integrity* (DFI) [27], can also detect data-only attacks. However, DFI imposes high overheads and is less commonly used, so here we focus on CFI.

Since the original introduction of CFI [24], more than a decade ago, a significant amount of research was devoted to proposing new schemes. However, there was not a similar amount of research on metrics to evaluate CFI schemes: most evaluation metrics, like *AIR* [52], are too simplistic and, as a result, they are unfit to properly describe the relative security guarantees of different CFI schemes.

CFINSIGHT With *CFInsight* [7, Appendix G], we introduce a new metric that is based on numeric properties of the program's *Control-Flow Graph* (CFG). Our metric is based on the observation that the attacker needs a system call invocation in order to access files or interact with the rest of the system. Hence, we base our metric on numeric properties of the paths leading to system call invocations. Moreover, we also introduce a new CFI policy generator that offers security guarantees similar to existing policy generators, and even better security when combined with existing generators.

1.3 MEMORY ISOLATION AND TRUSTED EXECUTION ENVIRONMENTS

A third approach to mitigate memory corruption is to isolate the various components of an application such that a vulnerability in a component cannot be leveraged to exploit a different component. Intuitively, memory isolation cannot prevent attacks within a component, but it is a useful technique to reduce the attack surface, especially when multiple untrusted components collaborate in a complex application.

A widely-deployed memory-isolation technique is protected memory: as an example, in Figure 1, Application A cannot access memory belonging to Application B. The same principle can be applied to isolate every logical component of the application. Application B in Figure 1 is divided into compartments ❸ such that a vulnerability in one of them cannot be used to attack another. *Software Fault Isolation* is an implementation of this idea in software [21, 28, 33].

However, the idea of protecting a sensitive component from others has also been implemented with the assistance of hardware extensions, namely in *Trusted Execution Environments* (TEEs). TEEs are designed to be effective even against a malicious operating system, which is a very powerful adversary. Application C in Figure 1 protects a sensitive component ❹ in a hardware-supported TEE.

Widespread TEEs include *TrustZone* (TZ) by ARM [59], *Software Guard Extensions* (SGX) by Intel [85], and *Secure Encrypted Virtualization* (SEV) by AMD [96].

While they differ on a number of details, the purpose of all TEEs is to ensure the confidentiality and integrity of a software component, protecting it from unauthorized access from any other component.

Memory isolation is particularly beneficial for software that is composed of a significant number of diverse components, especially if some of them handle untrusted data.

JITGUARD In *JITGuard*, we leverage a TEE to isolate a component of a web browser. Web browsers are very complex, even when compared with other modern applications. Due to this complexity and to the fact that they have to handle untrusted data and code, browser vulnerabilities are discovered on a regular basis. Additionally, browsers handle personal data, authentication credentials, and payment details, so they are very enticing targets for adversaries. A component that is particularly valuable for the attacker is the *Just-In-Time* (JIT) *compiler*, since it is able to generate new executable code. If the attacker is able to control the JIT compiler, the attacker can force it to emit malicious executable code directly into the browser process, as demonstrated by the data-only attack DOJITA [2, Appendix B]. With JITGuard [2, Appendix B], we leverage Intel SGX to isolate the JIT compiler from the other components and prevent any corruption of the compiler's internal data.

Memory isolation, and TEEs in particular, can be used for a number of purposes, including to ensure the confidentiality of a component. However, there is a category of attacks that is particularly effective in breaking the confidentiality property of memory isolation. These attacks leverage *side channels* in order to extract information from a component without directly interacting with it. Next, we examine these attacks and propose countermeasures.

1.4 SIDE-CHANNEL ATTACKS AND DEFENSES

Side-channel attacks represent a very stealthy threat. They manage to disclose information from a process without interacting with it directly: they monitor so-called *side channels*, i.e., unintended consequences of the execution of a program, in order to infer parts of the internal state of the victim. As an example, an adversary can leverage side channels against a cryptographic algorithm in order to extract a secret key.

The most common side channel happens through processor caches. While a program runs, it accesses its data, which is stored in the main memory, or RAM (❺ in Figure 1). In order to make future accesses faster, the data is stored in shared caches ❻ and core-private caches ❼. A malicious process running concurrently to the victim process can probe these caches and collect information through the side-channel leakage.

While side-channel attacks can be performed in a number of different settings, they are particularly effective against TEEs [89, 98]. Intel SGX in particular does not address side channels by design; instead, it is up to each protected component, called *enclave* in SGX parlance, to limit side-channel leakage.

While it is possible to use leakage-resistant and constant-time programming techniques, they require substantial manual effort from security experts. Hence, we propose both a software-only and a hardware-assisted technique to mitigate cache side-channel leakage without the need for manual effort.

DR.SGX With *DR.SGX* [4, Appendix D], we aim at mitigating the most direct source of cache-based side-channel leakage, i.e., the effects on cache produced by data accesses. DR.SGX introduces a software-only data permutation scheme, such that all data inside the enclave is periodically relocated

to a randomized location. As a result, the adversary does not know where the data is in the enclave; correlating cache-based side-channel signal with secret accesses performed by the enclave becomes much harder. Thanks to an efficient permutation scheme based on small-domain encryption, we can efficiently store the permutation inside the enclave without the need for a large table.

The root cause of cache-based side channels is shared caches. An expensive solution, albeit effective, is to partition the caches, so that they become virtually private. However, this quickly becomes impractical, especially if multiple mutually-untrusted enclaves are to run concurrently.

HYBCACHE *HybCache* [6, Appendix F] addresses the leakage problem in a different way: instead of partitioning the cache between trust domains, it enables mutually-untrusted domains to access a subset of the cache in a fully-associative fashion instead of set-associatively. With ordinary set-associative caches, if the adversary notices that a cache line has been evicted, the attacker gains information about which line the victim accessed. With a fully-associative access scheme, the adversary gains no information besides the fact that the victim accessed *a* cache line. Moreover, HybCache allows security-insensitive programs, executing outside of an enclave, to access the cache in the usual set-associative way, without any performance overhead.

In addition to proposing novel ways to mitigate side-channel leakage, we also set out to leverage TEEs in order to improve the performance and privacy of a number of distributed computation protocols.

1.5 IMPROVING PROTOCOLS WITH TEES

Ensuring trust and privacy in a distributed protocol without resorting to a trusted third party is challenging. As an example, this is the case with private execution of smart contracts over a cryptocurrency's blockchain. A general solution to this problem is *Secure Multi-Party Computation* (MPC), which comprises a number of cryptographic techniques that allow mutually distrusting parties to perform a computation without disclosing the inputs and the intermediate results to any of the parties. However, MPC techniques are very expensive in terms of computation time, network overhead, and financial burdens.

TEEs can be leveraged to implement similar protocols while greatly reducing the amount of cryptography required, since TEEs are designed to protect the confidentiality of data[4]. The introduction of a mutually trusted party in the form of a TEE allows the computation to be performed in a straightforward manner.

Similarly, the deployment of a TEE can introduce privacy guarantees in the domain of voice recognition, where most users have no choice but to trust the vendor not to abuse the data it collects. Below we describe our contributions in both of these domains.

FASTKITTEN The advent of cryptocurrencies gave birth to the idea of a *smart contract*, i.e., a secure and trustworthy way to run code according to a specified set of inputs in response to a certain condition. The most well-known smart contract platform is part of the cryptocurrency Ethereum. While Ethereum made smart contracts accessible to a large community, it does not

4 As we discussed in Section 1.4, most industry implementations of the TEE concept can be attacked using side-channel attacks; we also discuss our proposals to mitigate the possibility of these attacks. In this Section, we consider an idealized TEE that can withstand such attacks.

offer good support for code that is complex or that requires confidentiality of data. This is because Ethereum requires a large number of nodes to run the code of every contract every time it is executed. Moreover, it does not provide means to keep data private without complex and expensive cryptography. *FastKitten* [5, Appendix E] leverages a TEE to execute the smart contract in a protected environment and introduces a protocol that ensures that the participants and the TEE operator are treated fairly.

VOICEGUARD Voice recognition software is challenging to deploy in a privacy-preserving way. The main issue is that voice recognition needs both a voice recording and a machine learning model: the former belongs to the user and should ideally not be disclosed to the vendor, while the latter belongs to the vendor and should not be disclosed to competitors. The most common solution is to run the recognition software on the vendor's servers, leaving the users no choice besides simply trusting the vendor to respect their privacy. With *VoiceGuard* [3, Appendix C], we design a protocol that leverages a TEE running on the vendor's server to protect the confidentiality of the voice recording during the recognition process on the vendor's servers.

1.6 SUMMARY OF MY CONTRIBUTIONS

Like most academic works, the publications presented in this dissertation would not have been possible without the collaboration and support of my co-authors. Below I detail my contributions for each paper.

For *Selfrando* [1, Appendix A], the design was a collaboration between Christopher Liebchen, Andrei Homescu, Per Larsen, and me. I was the main author of the implementation, starting from existing code for a different project. I also performed the integration of the code within the Tor build architecture and the performance evaluation. This paper was developed in parallel with my master thesis [8]. While the thesis focuses on randomization and contains more technical details, the paper focuses more on defending against de-anonymization attacks on the Tor Browser and contains a new security analysis.

For *JITGuard* [2, Appendix B], I was the lead author of the design of the defense, in collaboration with Christopher Liebchen. I implemented most of the defense (except a component written by David Gens) and performed the evaluation. Christopher Liebchen developed and implemented the attack DOJITA.

In *VoiceGuard* [3, Appendix C], Ferdinand Brasser led the design with the collaboration of all authors. I contributed the proof-of-concept implementation and its evaluation, with the assistance of Korbinian Riedhammer regarding voice recognition systems.

For *DR.SGX* [4, Appendix D], the idea was developed jointly by Ferdinand Brasser, Alexandra Dmitrienko, Kari Kostiainen, and me, while I led the technical design. I also implemented the system and performed the evaluation.

For *FastKitten* [5, Appendix E], the idea is the fruit of the collaboration of all authors. Lisa Eckey and Kristina Hostáková designed the protocol and its security proofs. I led the implementation and performance evaluation of the prototype.

For *HybCache* [6, Appendix F] I contributed to the design discussions with Ghada Dessouky. I also performed the performance evaluation using an architectural simulator I implemented starting from preliminary work of a bachelor student.

Lastly, for *CFInsight* [7, Appendix G] I led the whole project, including design, implementation and evaluation, with important feedback from Patrick Jauernig and David Koisser.

2

MEMORY-CORRUPTION ATTACK MITIGATIONS

As mentioned in Chapter 1, memory corruption vulnerabilities allow the adversary to access memory in a way that was not intended by the programmer. Such vulnerabilities are particularly common in legacy programs written in memory-unsafe languages. These languages, including C and C++, require the programmer to manage memory manually, which is error-prone. As an example, when iterating over an array in C, there is no built-in construct that constrains memory accesses only to the array. Hence, if the programmer neglects to put some form of boundary check in place, this unconstrained memory access can be leveraged by the attacker to mount a memory-corruption attack. The goals of the adversary usually include reading and exfiltrating some information, overwriting some of the program's data, or completely hijacking the program to perform arbitrary computation.

The simplest way to achieve arbitrary computation with a memory-corruption attack is through a *code-injection attack*, in which the adversary simply injects some new code in memory and then redirects the control flow to the new code. To counter such attacks, a new defense was widely deployed, named W⊕X, which enforces the property that any memory page can either be writable or executable, but not both; hence, the adversary cannot inject any external code into executable memory.

As a result of the impossibility of directly injecting new code, adversaries started resorting to *code-reuse attacks*, which aim to use existing code in an unintended way to mount an attack of their choosing. Code reuse can be performed at various levels: the adversary can simply redirect execution to an existing function (e.g., in a *return-to-libc attack*), or can assemble the desired functionality using a number of very short code sequences (e.g., in a *Return-Oriented Programming* (ROP) [29] attack).

As we mention in Chapter 1, there are various approaches to limit the effectiveness of these attacks, including *software diversity*, *memory isolation*, and *integrity checks*. We discuss them in the following Sections.

2.1 SOFTWARE DIVERSITY

The core principle of software-diversity approaches is that, in order to transfer control to the building blocks of a code-reuse attack, the adversary needs to know the address in memory of each of these building blocks. Hence, the idea is to prevent the adversary from learning the address of any specific piece of code in the victim process.

2.1.1 *Our Contribution*

In the following paper:

[1] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi. "Selfrando: Securing the Tor Browser Against De-anonymization Exploits." In: *Privacy Enhancing Technologies Symposium*. 2016. CORE2021 rank: A, CORE2014 rank: B. Included in Appendix A on Page 45.

we present a load-time code randomization tool, *Selfrando*. Our tool acts at load time, i.e., after the protected application is loaded from disk but before

it starts executing. *Selfrando* randomly permutes the order of functions in memory: after the permutation, the application's code as a whole is still in the same memory region, but the order of functions is different at every run. Since the permutation only happens at load time, it has a minimal impact on run-time performance (less than 1% on SPEC CPU2006 benchmarks).

Our tool is completely self-contained, so it can run in a standard environment. It does not depend on a custom compiler, so it can be integrated in an existing build setup. It can also scale up to very complex codebases, including a web browser. As a result, we successfully collaborated with members of the Tor Project to prove the compatibility of *Selfrando* with their anonymous web browser, Tor Browser. Selfrando was also included in the hardened version of Tor Browser for Linux [67].

2.1.2 *Related Work*

ASLR The most widespread software-diversity approach is *Address Space Layout Randomization* (ASLR), which is nowadays commonly deployed on most commodity operating systems. It was first introduced in 2003 by the PaX Team [23] and it consists of shifting a number of memory areas, e.g., executable code, stack, heap, and libraries, by a random offset. ASLR was a very important first step against memory-corruption exploits, since it made code-reuse attacks harder and had very little performance overhead. However, ASLR has significant limitations. First, it suffers from low entropy on certain platforms (e.g., only 9 bits on 32-bit Linux [1, Appendix A]); *Selfrando* offers higher entropy. Second, if the attacker can disclose a pointer into a memory area (e.g., the executable code), the attacker can compute the address of any other portion of that memory area using trivial arithmetic. Since *Selfrando* shuffles memory on a function granularity, any pointer the adversary manages to leak only gives information about that function, but it does not give any information about the location of the other functions. Hence, the amount of information that can be gained from a pointer leak is substantially more limited.

FINE-GRAINED RANDOMIZATION Over the years, a number of works concentrated on the problem of constructing a randomization scheme which is more fine-grained than ASLR.

Bhatkar et al. [26] propose a randomization scheme based on a source-to-source transformation of C code. However, it suffers from a significant overhead of 11% and only supports C.

ILR by Hiser et al. [38] is based on a virtual machine that interprets custom binaries that use randomized instruction encodings. However, it suffers from an even more significant overhead of 15% and it is not compatible with JIT-compiled code, which is used, e.g., in web browsers to speed up execution of JavaScript programs.

Giuffrida et al. [36] propose a compiler-based approach that periodically re-randomizes the code. The downside of this approach is that it requires a custom compiler and build infrastructure to be deployed on the end user's machine, which would hamper adoption by non-technical users.

Binary stirring by Wartell et al. [40] is a binary-only approach that randomizes programs at install time, without needing the source code. Unfortunately, binary-only approaches require disassembling the binary, which is an imperfect process. As a result, this tool requires a run-time component to handle disassembly mistakes. It also relies on a commercial disassembler, which would severely complicate a real-world deployment.

Marlin by Gupta et al. [44] is a more lightweight binary-only approach; however, it can only handle simple binaries that disassemble without errors.

XIFER by Davi et al. [43] is a binary-only load-time randomization tool. Its downside is that it takes more than one second to randomize one megabyte of code: as a result, processing a modern web browser would take multiple minutes. Since processing takes place at load time, i.e., while the user is waiting for the application to start, this solution is not practical for complex software of this size.

Priyadarshan et al. [146] is another recently-proposed binary-only approach. They achieve a run time overhead of only 2.26%. However, this is still substantially higher than *Selfrando*'s overhead.

Koo et al. [128] propose a hybrid approach, which leverages rich information gathered at compile time and performs randomization at load time. It has an overhead which is comparable to *Selfrando*; however, it requires a client-side component, which limits the ability to deploy it in practice without support from the operating system vendor.

LIMITATIONS OF RANDOMIZATION Snow et al. [49] show that, if the attacker has access to an arbitrary read vulnerability, randomized code can be simply read and disassembled by the attacker, who can then assemble a custom exploit. This is possible regardless of the randomization granularity. Moreover, Bittau et al. [54] showed that the read vulnerability is not even necessary in some cases, if the adversary can observe whether the program crashed or not when trying a specific input. As a result, a number of research works [53, 63, 64, 79] have explored execute-only memory, which aims to prevent memory-disclosure attacks in the first place. Recent hardware features also enable efficient execute-only memory [61]. These execute-only memory solutions still require a code randomization tool like *Selfrando* to randomize the binary while they protect against memory-disclosure attacks.

2.2 MEMORY ISOLATION

The second approach we examine against memory-corruption attacks is *memory isolation*, which consists of introducing hardware or software barriers between different components such that a vulnerability in one component cannot be used to access a different component. Memory isolation can be applied in a number of ways.

In the *sandbox* model, a component which is subject to higher risk of compromise, e.g., because it processes untrusted data, is denied access to the rest of the system, except for well-defined interfaces, with the aim of containing a possible compromise inside the component.

Conversely, in the *enclave* model, a component that is particularly interesting for the attacker to compromise, e.g., because it contains sensitive data or encryption keys, is protected from the rest of the system such that other components can only access it through well-defined interfaces.

Below we show how we can apply the latter model to protect a sensitive component of a web browser.

2.2.1 *Our Contribution*

As we mentioned, modern web browsers are very complex and are composed of a high number of components. Web applications are also increasing in size and complexity. In order to run these applications with acceptable performance, web browsers adopt a number of techniques, including *Just-In-Time* (JIT) compilation of JavaScript code. With this technique, a browser automatically identifies JavaScript functions that are executed often and compiles them into native code, which can be directly executed by the hardware

without the need for a JavaScript interpreter. After a small initial performance penalty due to the compilation, execution speed of native code is significantly faster than a traditional interpreter.

However, this process introduces the possibility of inserting additional code into executable memory. If the adversary can read and write arbitrary memory through a vulnerability, it is possible to hijack the JIT compilation process and produce malicious executable code, which the adversary can later call [2, Appendix B].

In the following paper:

[2] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi. "JITGuard: Hardening Just-in-time Compilers with SGX." in: *ACM Conference on Computer and Communications Security (CCS)*. 2017. CORE2021 rank: A∗. Included in Appendix B on Page 61.

we introduce *JITGuard*, a memory-isolation defense that protects the integrity of the JIT compilation process.

JITGuard protects the JIT compiler inside an Intel SGX enclave, making it impossible for the attacker to interfere with its data. However, the JIT-compiled code exists outside of the SGX enclave, in order to allow fast switches to and from static code. Moreover, *JITGuard* protects JIT-compiled code from code-injection and code-reuse attacks by leveraging a data hiding scheme. The location of the JIT-compiled code is protected by trampolines and data scrubbing, while efficient code updates are allowed through a second memory mapping, which is writable and randomized. Our proof-of-concept implementation protects Firefox's JavaScript engine and shows an overhead of 9.8%.

2.2.2 *Related Work*

CODE-INJECTION ATTACKS AND DEFENSES FOR JIT COMPILERS JIT compilers change the JIT-compiled code often, e.g., to add a new function, to optimize existing code, or to remove code that is no longer needed. As a result, memory permissions of memory pages containing JIT code were traditionally set to RWX, i.e., both execution and modification were enabled at the same time. While convenient, this setup is a violation of the W⊕X policy. An attacker could simply inject new code and run it [83].

JIT compilers that respect the W⊕X policy have been proposed in the academic world [35, 42] and a similar approach has been deployed in commercial browsers as well [70].

However, these schemes require switching the memory from executable to writable when the code needs to be modified. This brief window when the code is writable can still be exploited, as shown by Song et al. [76]. As a mitigation, they propose to move the compiler to a separate process, so that the executable memory is never writable in the main browser process. While effective, this defense requires a tremendous amount of *Inter-Process Communication* (IPC), leading to overheads as high as 50%. A similar approach has been deployed in Microsoft Edge [84, 111]. A downside of this effort is that it breaks the synchronous call semantics of existing JIT-enabled JavaScript engines, since code modification relies on asynchronous IPC instead. As a result, adapting a JIT compiler to this model is a "non-trivial engineering task" [111].

In contrast to the separate-process approach, interactions with the SGX enclave used in *JITGuard* preserve conventional synchronous call semantics, leading to an easier adaptation task.

CODE-REUSE ATTACKS AND DEFENSES FOR JIT COMPILERS Even if the adversary is successfully prevented from injecting new code, the adversary still has another opportunity to influence JIT-compiled code: by forcing the compiler to emit malicious code fragments, e.g., by embedding it into constants [31]. For more details about these attacks, we refer the reader to the SoK paper by Gawlik et al. [123].

A possible defense approach is to monitor system calls originating from JIT-compiled code and reject them [32]; however, this only works if benign JIT-compiled code never contains system calls, and does not protect against code-reuse attacks that do not directly invoke system calls.

A different approach is *constant blinding* [31], which consists of storing constants xor-ed against a random value, so that attacker-controlled constants only exist in registers but not in memory. However, for performance reasons, this approach is usually not applied to small constants [109], which can be still exploited by the adversary [60].

Yet another defense technique is to apply code randomization to JIT-compiled code [45]; however, code randomization is vulnerable to memory disclosure attacks, as we discuss in more detail in Section 2.1.2.

Control-Flow Integrity (CFI) can be leveraged too [57], with an overhead of 14%. This approach does not detect attacks that do not deviate from the expected *Control-Flow Graph* (CFG), e.g., data-only attacks.

Software Fault Isolation (SFI) can be applied to JIT-compiled code as well; however, this can lead to overheads greater than 20% [34].

Lastly, NoJITsu by Park et al. [145] builds on the *JITGuard* design to construct an even more comprehensive defense that protects the JavaScript interpreter as well as the JIT compiler. However, this defense leverages and requires Intel's *Memory Protection Keys* (MPK) hardware extension. MPK was not available at the time *JITGuard* was designed and is still only available on a limited number of processor models. *JITGuard* builds on SGX instead, which is widely available on recent Intel processors.

IMPACT OF SGX DATA LEAKAGE ATTACKS As we explain in more details in Section 3.1.2, a number of side-channel data-leakage attacks have been demonstrated against SGX. However, most of these attacks undermine the *confidentiality* of SGX, not its integrity.[1] In *JITGuard*, the adversary needs to *alter* the internal data of the compiler, but gains no advantage by simply *knowing* it. The only exception is the location of the secret randomized memory region; however, it would be challenging for the attacker to leak it, since our adversary has no access to cache-related and other low-level instructions (unlike the standard SGX threat model, where the host application and the operating system may be under control of the adversary).

2.3 CONTROL-FLOW INTEGRITY

The core idea of *Control-Flow Integrity* (CFI) is to make sure function call sites can only call intended call targets, while unintended calls are blocked. This way, the program's control flow is not compromised by the adversary, i.e., its integrity is preserved.

A CFI scheme requires various components. The *enforcement* component has the responsibility of deciding whether or not to block a particular call at run time, based on a CFI *policy*. The CFI policy, produced by a *policy generator*, describes which calls are allowed and which are not, based on different criteria.

1 Some attacks have been proposed that introduce faults in SGX enclaves [19, 143], thus undermining the integrity of the enclave. However, making the JIT compiler malfunction (and probably crash) does not give our adversary any advantage.

CFI policies can be defined at various levels of granularity. As an example, a simple coarse-grained CFI policy is to allow every call site to call any function. While simple, this policy already blocks any call into the middle of a function, hence severely complicating ROP attacks. However, full-function reuse attacks are still possible. In order to mitigate them as well, finer-grained CFI policies can be defined.

A general approach to generate a CFI policy is to start from the *Control-Flow Graph* (CFG) of the program to protect. A CFG is a graph containing all basic blocks of a program as nodes, and all expected control flow transfers as edges. Extracting a precise CFG of a program is a well-known problem; however, it is possible to produce approximated CFGs.

Having a CFG, a simple CFI policy is to allow a control flow transfer if and only if the corresponding edge is present in the CFG. This constraint can also be relaxed for performance reasons: a common optimization is to assign every function a numeric label and only allow each call site to call functions with one specific label. The downside of this approach is that any function callable from the same call site needs to have the same label, leading to a loss in precision. Another approach, often deployed in the real world, is to check that the function type signature of the callee matches the one expected at the call site; a simplified variant is to just check the number of arguments, without considering their types.

Given this wide variety of CFI approaches, it is important to be able to compare them in terms of various properties, including performance impact and security guarantees. Measuring the performance impact of a CFI implementation can be performed in a straightforward way by comparing the run time of a benchmark suite with and without the CFI protection. However, defining a CFI security metric is not trivial. The most commonly-used metric, *Average Indirect-target Reduction* (AIR) [52], is defined as the mean of the ratio between the number of allowed call targets and the total possible targets. However, AIR is not an effective CFI security metric, for two reasons. First, most CFI implementations report similar AIR values higher than 99% [99], which complicates a direct comparison. Second, even CFI schemes with very high AIR have been shown to be vulnerable to attacks [55, 56]. Hence, AIR is not an adequate means to measure the security of CFI variants. We mention other CFI metrics and their shortcomings in Section 2.3.2.

2.3.1 *Our Contribution*

In the following paper:

[7] T. Frassetto, P. Jauernig, D. Koisser, and A.-R. Sadeghi. "CFInsight: A Comprehensive Metric for CFI Policies." In: *Network and Distributed System Security Symposium (NDSS)*. 2022. CORE2021 rank: A∗. Included in Appendix G on Page 133.

we start from the observation that not all basic blocks are useful to the adversary. Since a pure ROP attack is mitigated by even coarse-grained CFI, the best path the adversary has to arbitrary code execution is to invoke a system call and change the memory permissions. Even if the adversary wants to attack the operating system or exfiltrate some files, one or more system calls are required. Hence, we introduce a new CFI metric, named CFGInsulation, which is based on the assumption that the adversary needs to reach a system call instruction. Our metric is formulated such that a program is more secure if the distance to a system call instruction is higher. Additionally, the program is less secure if there are many independent paths leading to the system call

instruction. By computing this metric for a program protected by various CFI variants (as well as unprotected) we can compare their security effectiveness.

Moreover, we define a new CFI policy generator, named NumCFI, which enforces the property that the distance to a system call instruction can decrease at most by 1 at every basic block transition. As a result, the adversary is prevented from taking "shortcuts" to reach a system call. We show how NumCFI is effective both on its own and combined with other existing CFI policy generators.

Finally, we show a proof-of-concept implementation of a CFI enforcement mechanism capable of enforcing a NumCFI policy, with a run time overhead of 1.27%.

2.3.2 *Related Work*

CFI SCHEMES In 2005, Abadi et al. [25] introduce the concept of CFI. Their design is based on *labels*, just like many later CFI variants. As we mentioned earlier, a label-based CFI approach assigns a single numeric label to every possible call target, then instruments indirect calls so that the actual label of the callee is checked against the expected label value. This effectively splits callers and callees in equivalence classes, one for each possible label value. Our novel policy generator NumCFI is orthogonal to label-based approaches, as it is based on an ordering relation instead of a equivalence relation. The same paper [25] also introduces the first CFI implementation, which uses just one label for all address-taken functions. It instruments binaries using a proprietary instrumentation tool.

Zhang et al. [52] propose a binary-only instrumentation tool, without the need to have any access to the build process. Zhang et al. [51] propose a randomized CFI approach which leverages a "springboard section"; indirect control transfers need to know the correct entry for the intended callee in the springboard. Tice et al. [58] extend this technique by also protecting virtual C++ calls. Lockdown by Payer et al. [73] dynamically generates CFI checks at run time.

A number of research efforts leverage function parameter type information to restrict control-flow transfers. TypeArmor by Van Der Veen et al. [93] extracts the number of parameters of each function using binary analysis, then instruments code to only allow function calls if enough arguments are passed. RAP by the PaX Team [72] and τCFI by Muntean et al. [129] also consider the parameter types in order to decide whether to allow a control-flow transfer. MARX by Pawlowski et al. [113] and VCI by Elsabagh et al. [103] further refine the type system by taking into account C++-specific virtual calls based on virtual tables. Clang, the front-end of the LLVM project, also includes a type-based CFI protection based on the dynamic types of variables. In general, type-based CFI implementations tend to have a low overhead and good compatibility. Hence, they are often deployed in the real world. However, they do not consider the distance of basic blocks to system calls or other attacker targets, so that some attacks are still possible [122].

Another research direction is to integrate additional contextual information into the CFI policy. As an example, a *context-sensitive* CFI policy considers not only the basic block that contains the call site, but also the call stack [77, 102, 125]. *CFInsight* does not support context-sensitive CFI in the current form, but the approach can be applied to model context-sensitive policies as well.

EXISTING CFI SECURITY METRICS As we already mentioned, security metrics for CFI are important tools to compare the effectiveness of various CFI schemes.

The most widespread CFI security metric is *Average Indirect-target Reduction* (AIR), which was introduced in 2013 by Zhang et al. [52]. In order to compute AIR, one must consider every indirect control transfer instruction; specifically, how many targets are allowed by the CFI policy and how many targets are possible without CFI. AIR is then defined as the mean of the ratio between the allowed targets and the possible targets. Similarly, AIA by Ge et al. [82] is defined as the average number of allowed targets, without considering the total number of possible targets. iCTR by Muntean et al. [139] is defined as the sum of the number of the allowed targets over all indirect control transfer instructions. Lastly, QuantitativeSecurity by Burow et al. [99] is defined as the ratio between the number of CFI equivalence classes and the size of the largest equivalence class. These metrics do not consider the specific position of a basic block inside the CFG and the block's connectivity; our metric CFGInsulation improves on this by considering the paths from basic blocks to system calls.

3 SIDE-CHANNEL ATTACK MITIGATIONS

As we mentioned, memory-corruption attacks leverage vulnerabilities in a program to subvert its behavior. *Side-channel* attacks are even more devious, as they allow the adversary to extract information even from a bug-free program.

Side-channel attacks are only possible if both the attacker and the victim can access the same resource, e.g., a shared cache or buffer. In general, a side-channel attack requires three steps. First, the adversary sets up the shared resource so that it is in a known state. Second, the victim executes for some amount of time. Third, the adversary examines the state of the shared resource, compares it with the previous known state, and extracts secret information from the difference in state.

We will focus on cache-based side-channel attacks. On modern computers, caches are used to provide lower-latency access to recently-used data. All data in main memory is divided into *cache lines*; when the processor needs to access some data, it requests the corresponding cache line(s) from the cache. If the data is available in the cache, it is promptly returned to the processor, otherwise it is fetched from the memory. Multiple levels of caches are possible and commonplace.

Most modern shared caches are accessed in a *set-associative* way. This means that the cache is divided in a number of *sets*; each cache line can only be stored into a specific set, depending on its address in main memory. In this context, the attacker can perform a simple cache-based side-channel attack by filling the entire cache, letting the victim execute, then checking how many of the attacker's cache lines are still in cache. Each cache line of the attacker that is no longer in the cache was evicted by a cache line of the victim. Hence, if a line of the attacker was evicted, the attacker knows that the victim accessed a cache line within the same cache set. This knowledge about (partial) addresses accessed by the victim can be leveraged in a number of ways, but it is especially powerful if the victim executes some cryptographic operation and the memory accesses depend on a secret key.

3.1 SOFTWARE-ONLY SIDE-CHANNEL MITIGATIONS

Cache policies are implemented by hardware. In most cases, independent software vendors design their code for a commodity hardware platform, which cannot be changed. If the hardware is fixed, the algorithm can be implemented in such a way that side-channel leakage is minimized. It is possible to change the algorithm manually (e.g., by accessing all items of an array instead of only accessing the one required variable) so that no discernible pattern exists in the victim's memory accesses. However, side-channel-resilient implementations require a substantial amount of manual effort by security experts, while still being error-prone. Hence, we designed a tool capable of automatically protecting access to data, without the need for manual analysis.

3.1.1 *Our Contribution*

Cache-based side-channel attacks are particularly relevant on SGX, as such attacks are purposefully excluded from the official SGX thread model. To mitigate these attacks, in the following paper:

[4] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi. "DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization." In: *Annual Computer Security Applications Conference (ACSAC)*. 2019. CORE2021 rank: national USA, CORE2018 rank: A. Included in Appendix D on Page 83.

we design and implement an automated data randomization framework for SGX enclaves. Our approach is agnostic to the application semantics and does not require manual annotations. The data is randomized at the granularity of a cache line, thus mitigating both paging-based and caching-based side channels, and is periodically re-randomized to mitigate correlation attacks. Our evaluation shows that *DR.SGX* causes overheads between approximately 5× and 11×, depending on the re-randomization frequency, unlike approaches based on *Oblivious RAM* (ORAM), which have overheads in the order of 100× [135].

3.1.2 *Related Work*

SIDE-CHANNEL ATTACKS ON TEES In SGX, memory management, including paging, is managed by the operating system [80], which is considered untrusted. As a result, the operating system can very easily build a noise-free side channel based on paging [78, 119], which leaks the address of every memory access with the granularity of a page (4 KB). In addition to the paging-based side channel, an adversary can also leverage leakage based on a caching-based side channel. A cache-based side channel is subject to noise, but it can leak more fine-grained information than paging [98, 106, 112, 115, 124]. Combining leakage based on paging and caching is possible, which leaks the address of memory accesses with the granularity of a cache line (64 bytes).

TRANSIENT EXECUTION ATTACKS ON TEES In recent years, a number of attacks have been proposed that leverage transient execution vulnerabilities, e.g., Foreshadow [134]. These attacks leverage side channels that had not been demonstrated previously; Intel patched them in microcode updates and fixed them in hardware for later products [120]. *DR.SGX* focuses instead on cache-based and paging-based side channels, which are still possible on current hardware.

SIDE-CHANNEL DEFENSES One of the best options for the adversary is to deploy the attack code on the logical partner of the victim core, so that both can access the same L1 cache. This is only possible if *Simultaneous Multi-Threading* (SMT) is supported and enabled on the processor. As a mitigation, Intel introduced the possibility to disable this feature, and included this setting in the attestation report [137], in order to allow remote parties to only trust enclaves that run without SMT. Varys by Oleksenko et al. [130] achieves the same goal by running two threads at the same time and periodically checking that they are executing on two logical partner cores. However, it does not protect the *Last-Level Cache* (LLC), which can be accessed by other physical cores. Moreover, blocking two logical cores or even disabling SMT altogether limits the processing power of the machine and does not protect against the paging-based side channel. In contrast, with *DR.SGX* all other logical cores remain available.

A number of research works leverage ORAM to protect execution in a TEE. ORAM [20, 22, 37, 41, 50, 75] was originally developed in a client-server environment and allows *oblivious* accesses to memory, i.e., an adversary that monitors traffic cannot learn which data is being accessed. *Oblivious execution* enhances ORAM by also preventing the adversary from learning what pro-

gram instructions are executed and which control flow paths are followed [46, 47, 69]. Obfuscuro by Ahmad et al. [135] is an oblivious execution implementation for SGX. The overhead of full oblivious execution on SGX is significant, with an average of 83× and peaks of 220×. In contrast, *DR.SGX*'s overhead is lower by an order of magnitude.

Sinha et al. [118] automatically protect a program written in a custom language from paging-based side channels. However, unlike *DR.SGX*, it does not support legacy code.

Raccoon by Rane et al. [74] provides targeted protection for specific data, relying on developer annotations. Protected data is then accessed obliviously. Similarly, CoSMIX by Orenbach et al. [140] is a compiler-based framework that identifies sensitive data based on annotations and static analysis. *DR.SGX* does not require developer annotations.

A different line of research aims to develop leakage-resilient algorithms and application-specific defenses, e.g., for machine learning algorithms [92], for the MapReduce framework [71], for a database [104], or for encrypted search [121]. Application-specific defenses can be very effective in the specific case but require substantial effort to be designed and implemented; *DR.SGX* strives to be generally applicable with limited additional effort.

A number of works focus on detecting attacks that require frequent interruptions: T-SGX by Shih et al. [117] leverages Intel's *Transactional Synchronization Extensions* (TSX), while Déjá Vu by Chen et al. [100] monitors the execution time.

Cloak by Gruss et al. [107] also uses TSX to touch a number of cache lines before sensitive memory accesses to developer-annotated code regions, making them atomic. Due to the complexity of loading large amounts of data in the cache, Cloak can have overheads up to 30× in some cases. Moreover, these approaches can only be deployed on processors that support TSX.

Autarky [144] mitigates the paging-based side channels by allowing the enclave to control paging; however, it requires hardware modifications.

Code diversity approaches [62] can also be deployed in the context of a TEE, complementing the data randomization proposed in *DR.SGX*. As an example, SGXShield by Seo et al. [116] demonstrates code randomization for SGX; however, its implementation is incomplete [11].

NEW CACHE ARCHITECTURES A different approach to eliminate caching-based side-channel leakage at its root is to redesign the cache architecture with this goal in mind. We describe our own proposal in Section 3.2.1 and the related work in Section 3.2.2.

3.2 HARDWARE-ASSISTED SIDE-CHANNEL MITIGATIONS

In contrast to software-only side-channel mitigations, which we considered in the previous Section, *hardware-assisted* side-channel mitigations leverage one or more hardware security features to protect programs. On the one hand, due to the dependency on custom hardware features, these defenses are not easily deployable on existing platforms. On the other hand, hardware modifications can solve the problem at its root, providing enhanced security guarantees with limited performance overhead.

In the context of cache-based side channels, a simple and secure solution is to *partition* the cache, so that each program has exclusive use of a portion of the cache. However, cache partitioning has the substantial drawback that the number of possible partitions is limited and that each program gets an increasingly small amount of cache, leading to performance slowdowns. While these issues can be partially alleviated by assigning multiple programs to the same partitions, cache partitioning still imposes a significant overhead. This

is especially unfortunate if some of the programs are not sensitive and do not require protection at all.

3.2.1 *Our Contribution*

In order to address this problem, in the following paper:

[6] G. Dessouky, T. Frassetto, and A.-R. Sadeghi. "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments." In: *29th USENIX Security Symposium*. 2020. CORE2021 rank: A*. Included in Appendix F on Page 115.

we introduce *HybCache*, a hybrid cache design that combines a traditional set-associative cache and a leakage-resilient fully-associative *subcache*. The core observation at the base of *HybCache* is that only a *subset* of the programs on a system are security-sensitive and require protection against side channels; often, the majority of the executing code does not require protection. As an example, if the workload of a system is already split between a few trusted components (that run in TEEs) and the rest of the code, it is likely that only the trusted components require side-channel protection. The rest of the workload likely does not require protection.

Hence, *HybCache* caches are accessed differently, depending on whether the requester is security-sensitive or not. If the request comes from a security-insensitive program, the whole cache is accessed set-associatively, as usual. However, a portion of the cache, called *subcache*, can also be accessed fully-associatively by security-sensitive code. Moreover, security-sensitive requests use a random replacement policy. Since these access and replacement policies are independent from the address of the data accessed, it is impossible for an adversary to gain any information regarding which addresses the victim is accessing. While the adversary cannot learn anything about *which* memory addresses are accessed by the victim, a powerful adversary can still observe that the victim performed *some* memory accesses. While unfortunate, this leakage can only be solved by full cache partitioning, which has substantial performance overhead.

As a consequence of our design, security-insensitive cache accesses sustain *no* overhead in our evaluation. Security-sensitive accesses show overheads of 3.5%–5%, which is a small price considering the side-channel protection.

3.2.2 *Related Work*

CACHE PARTITIONING The core idea of cache partitioning is to divide the cache in multiple partitions so that different components of the workload (e.g., components or TEEs) cannot interfere with each other's cache lines, preventing cache-based side channels.

Cache partitioning can be realized by splitting the available cache in disjointed subsets, either through hardware modifications or using just software. An example of the latter is *page coloring*, a technique based on choosing the physical address of memory pages in such a way that different components use disjointed groups of cache sets.

A similar result can be achieved by enforcing a cache flush when a context switch between mutually distrusting components happens. Clearly, this is only applicable to private caches (which are only accessed by one component at a time).

Static cache partitioning can lead to slowdowns in the order of 40% even with only two partitions [94]; increasing the number of partitions decreases their size, leading to even higher overheads. SecDCP by Wang et al. [94] and

CacheBar by Zhou et al. [95] dynamically manage the size of each partition depending on each application's requirements to try and reduce the overhead. In Sanctum by Costan et al. [81], private caches are flushed on context switch, while shared caches are partitioned. Since the minimum size of partitions is fixed, they do not scale well to a higher number of partitions. The overhead also applies to security-insensitive code, unlike *HybCache* which does not slow it down. Chunked-Cache by Dessouky et al. [101] partitions shared caches based on more flexible chunks, unlike previous approaches that use way-based partitioning, allowing for better scalability.

A number of works partition the cache between security-sensitive and security-insensitive portions. StealthMem by Kim et al. [39] achieves this through page coloring, while CATalyst by Liu et al. [90] leverages Intel's *Cache Allocation Technology* (CAT). PLcache by Wang et al. [30] includes hardware modifications that allow sensitive processes to lock some lines into the cache so that they cannot be evicted by other processes. All of these solutions reduce the amount of cache which is usable by security-insensitive programs, unlike *HybCache*.

It is important to note that partitioning caches on commodity hardware requires the cooperation of the operating system. Hence, it is not suitable for user-space TEEs like SGX. We describe some TEE-compatible approaches in Section 3.1.2.

CACHE RANDOMIZATION A number of approaches have been proposed that randomize the mapping function from the memory address to the possible locations in cache. RPCache by Wang et al. [30] randomly reassigns cache lines from a cache set to another, while NewCache by Liu et al. [91] randomizes the mapping at the cache line level. Time-Secure Cache by Trilla et al. [133] uses a mapping function that depends on the ID of the requesting process. ScatterCache by Werner et al. [142] uses a different pseudo-random cache set mapping for each security domain. While these approaches obfuscate the mapping algorithm, they still have a deterministic mapping of cache lines to portions of the cache. As a result, an attacker can still correlate access patterns given enough repetitions, unless the mapping function is re-keyed, which imposes overheads.

Mirage by Saileshwar et al. [148] uses indirection to decouple the tag store from the data store, in order to allow for a fully-associative cache with lookup performance similar to set-associative caches. However, the performance overhead of this design applies to the whole workload, while *HybCache* does not impose overheads on non-security-sensitive code.

4 TEE APPLICATIONS

In the previous chapters we have proposed a number of system-level security mitigations against run-time attacks. A number of works have focused on leveraging TEEs for security purposes or protecting them from attacks. We now conclude with two examples of how TEEs can be used to significantly simplify the design of protocols.

In general, when mutually distrusting parties want to collaborate in a protocol, a simple solution is to resort to a *trusted third party*, which can collect inputs from all participants and then securely conduct the protocol. Often, though, finding a third party which is trusted by all participants is impractical or even impossible.

A general solution to this problem is *Secure Multi-Party Computation* (MPC), which is a cryptographic technique that allows mutually distrusting parties to perform a computation without disclosing the inputs and the intermediate results to any of the parties. However, MPC techniques are very expensive in terms of computation time and network overhead.

The use of TEEs allows for significant improvements in the performance compared to MPC approaches, without the need to trust anyone (besides the hardware manufacturer). TEEs can then be used similarly to a "programmable trusted third party", with the added advantage of remote attestation, through which every participant can get a confirmation that the TEE is executing the expected program.

As we mentioned in Chapter 3, TEEs can be vulnerable to a number of side-channel attacks; however, a number of mitigations have been proposed as well (Sections 3.1.1, 3.1.2, 3.2.1 and 3.2.2). In this Chapter we consider idealized TEEs that can withstand these attacks.

4.1 AUTOMATED SPEECH RECOGNITION

Automated speech recognition is an increasingly used technique, e.g., by a number of smart digital assistants. Speech recognition is usually performed through machine learning. Machine learning models are provided by a handful of vendors, which want to protect them from their competition and prefer to keep them on their own servers. However, these models require the recording of the user's voice in order to work. The voice recording often contains sensitive biometric data and user information.

Traditionally, the voice recognition process happens on the vendor's servers: this effectively protects the machine learning models, but the user has no guarantee on how the recording is processed. Conversely, performing the recognition directly on the user's device protects the recording, but leaves the models vulnerable.

4.1.1 *Our Contribution*

To address this problem, in the following paper:

[3] F. Brasser, T. Frassetto, K. Riedhammer, A.-R. Sadeghi, T. Schneider, and C. Weinert. "VoiceGuard: Secure and Private Speech Processing." In: *Interspeech*. 2018. CORE2021 rank: A. Included in Appendix C on Page 77.

we leverage an SGX enclave to securely process voice commands while protecting both the recording and the recognition models. The voice recognition algorithm, which does not contain secret information, is prepared by the vendor and made available to the users, who can inspect it and make sure that the recording is not disclosed. The vendor then prepares a TEE which executes the published code (users can verify this through remote attestation). The TEE receives, in encrypted form, the voice recognition models and the voice recording; it then processes the recording and only releases the textual representation of the user's command. Through this design, both parties can be sure that their data is kept confidential.

In our evaluation, we show that this design is realistic and that real-time processing of voice recordings is possible.

4.2 EFFICIENT SMART CONTRACTS ON LEGACY BLOCKCHAINS

Cryptocurrencies are prominent examples of decentralized systems, which do not rely on any central trusted party. Earlier examples, including Bitcoin, only support simple operations like transfer of funds. Later cryptocurrencies, like Ethereum, expanded their feature set to include *smart contracts*, i.e., pieces of code that can implement arbitrary computation. However, Ethereum imposes significant limitations and costs for smart contracts, which limit their possible complexity; Bitcoin does not support them at all.

4.2.1 *Our Contribution*

To address this problem, in the following paper:

[5] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi. "FastKitten: Practical Smart Contracts on Bitcoin." In: *28th USENIX Security Symposium*. 2019. CORE2021 rank: A*. Included in Appendix E on Page 97.

we propose *FastKitten*, which allows efficient execution of smart contracts in a TEE. Due to the flexibility and simplicity of programming a TEE, *FastKitten* can support complex smart contracts; their execution is efficient, since they can be executed just once instead of on multiple nodes. In most cases the execution does not interact with the blockchain, which allows real-time interaction between participants and the contract. Moreover, *FastKitten* can be deployed on top of simpler blockchains that do not support smart contracts; we show this by demonstrating a proof-of-concept implementation based on Bitcoin. Lastly, smart contracts in *FastKitten* keep their state private by design, unlike distributed smart contracts on Ethereum, whose data is inherently public.

4.3 RELATED WORK

CRYPTOGRAPHIC TECHNIQUES *Secure Multi-Party Computation* (MPC) is a class of cryptographic techniques that allow multiple parties to collaboratively perform a computation without disclosing the inputs or intermediate results to the parties. The most significant advantage of MPC is that it does not require a trusted third party: the protocol itself guarantees the data confidentiality. A number of solutions have been proposed that leverage MPC to evaluate machine learning models [108, 126, 131, 132], including for voice recognition [48, 105], and to run smart contracts interacting with a blockchain [68, 87, 88]. However, MPC imposes very high overheads in terms of networking messages, processing speed, and start-up times; the latter is

particularly unfortunate for real-time speech recognition, while the former two are challenging in any scenario.

TEES FOR MPC AND SPEECH RECOGNITION A logical step is to introduce TEEs into MPC protocols [66, 86, 97]. The introduction of TEEs decreases computation and networking overheads significantly; however, it requires trusting the platform vendor and its hardware, while traditional MPC only requires trusting the protocol.

After the publication of *VoiceGuard*, we also worked on Offline Model Guard [16], which runs speech recognition tasks on a user's device. Similarly to *VoiceGuard*, OMG protects the confidentiality of the speech recognition model using a TEE. However, since the model runs on the user's device, it can be used even without an active network connection.

TEE APPLICATIONS FOR SMART CONTRACTS INTERACTING WITH A BLOCKCHAIN The popular cryptocurrency platform Ethereum supports *smart contracts*, which are pieces of code that live on the blockchain and can interact with different (untrusted) users. Smart contracts are executed by a large number of nodes; hence, tampering with them is considered infeasible (except in the case of programming errors). While the existence of smart contracts allowed the creation of a wide variety of decentralized services, Ethereum contracts have some downsides. First and foremost, they are inherently public, along with their input data. Second, it is expensive to run complex smart contracts, due to the fact that they need to be run by a large number of nodes. Third, they are only possible on Ethereum; Bitcoin, another popular cryptocurrency, does not support complex smart contracts.

A common approach to address some of these challenges is to create so-called *second-layer solutions*, like state channels [110], Arbitrum [127] or Plasma [114]. These approaches have the disadvantage that, if the parties do not agree on the result of a smart contract, the dispute has to be resolved on the blockchain, which is expensive.

Similarly to *FastKitten*, Ekiden [136] leverages a TEE to run smart contracts off-chain. However, Ekiden only supports input from one client at a time and only receives inputs from the blockchain, leading to low throughput and high costs for interactive programs. With *FastKitten*, the whole program execution, possibly including multiple rounds of inputs, happens offline in the optimistic case. Hyperledger Avalon [141] is an industry proposal with downsides similar to Ekiden.

5 CONCLUSION AND OUTLOOK

Due to the rising complexity of computer systems and programs, ensuring that a program is bug-free is increasingly impractical. While manual and automated bug-finding strategies are important and beneficial, they are not sufficient. Hence, it is also important to develop vulnerability mitigation strategies, which aim to make the exploitation of these vulnerabilities significantly harder or even impossible.

We start our presentation in Chapter 2, in which we focus on defenses against run-time attacks.

Selfrando [1, Appendix A] is a load-time code randomization tool. It applies fine-grained randomization to existing applications every time they start, while allowing the use of traditional distribution channels and debugging tools. The use of randomization makes it significantly harder for the attacker to know the location of any piece of code in memory, which is required to mount memory-corruption attacks.

JITGuard [2, Appendix B] uses memory isolation to protect from tampering the internal data of a Web browser's JIT compiler. It leverages an SGX enclave to isolate the JIT compiler from unintended accesses. It also mitigates other attacks on JIT-compiled code by randomizing its location in memory and using a trampoline layer. JIT-compiled code resides outside of the enclave in order to preserve performance.

CFInsight [7, Appendix G] introduces new metrics to evaluate the security effectiveness of CFI policies. It considers all paths from possible memory corruption points to system call instructions to quantify how hard it is for the adversary to reach its goal. The same paper introduces a new CFI policy generator, NumCFI, which enforces the property that the distance to a system call instruction can decrease at most by 1 at a time. NumCFI prevents the attacker from "taking shortcuts" to a system call instruction.

In Chapter 3 we move our focus to a class of attacks which are even more devious: side-channel attacks on TEEs.

DR.SGX [4, Appendix D] is a software-only randomization tool for data inside of an SGX enclave. The location of all data inside the enclave is randomized in order to mitigate side-channel attacks that aim to understand the address where some variable is located. Periodic re-randomization is used when the enclave runs for an extended amount of time. The data permutation is stored efficiently leveraging small-domain encryption, while a side-channel-oblivious permutation cache improves performance.

HybCache [6, Appendix F] leverages a new cache paradigm to prevent a number of cache-based side-channel attacks at the root. Security-sensitive data requests are processed fully-associatively and with a random replacement policy in a special part of the cache. As a result, it is impossible to mount an attack that leaks the address of a variable. Security-insensitive data requests access the cache using the traditional set-associative scheme, and hence, their performance is unaffected.

Lastly, in Chapter 4 we present two applications of TEE technology.

In *VoiceGuard* [3, Appendix C] we show how a TEE can be used to perform automated voice recognition while protecting from disclosure both the machine learning model and the voice recording.

In *FastKitten* [5, Appendix E] we use a TEE to enable fast interactive execution of smart contracts on top of Bitcoin, a cryptocurrency that does not natively support complex smart contracts.

5.1 FUTURE DIRECTIONS

We have shown a number of defenses against control-flow hijacking, side-channel leakage, and data-only attacks on JIT compilers.

Due to the fast pace of the tech industry, security solutions and mitigations need to be periodically updated in light of new technologies. New industry proposals, e.g., AMD's *Secure Encrypted Virtualization* (SEV) and Intel's *Trust Domain Extensions* (TDX), allow new capabilities that can be exploited, but at the same have the potential to introduce new vulnerabilities as well.

At the same time, there are some common tasks, e.g., data hiding, which are very hard to implement on commodity platforms. It would be beneficial to have platform support for custom address spaces for security-sensitive data, e.g., through segmentation or custom instructions [13].

Moreover, while scientific publications often tend to have a very narrow focus, holistic approaches are also valuable: as an example, a comprehensive model of run-time attack possibilities in various conditions, in different programming paradigms and under one or more run-time mitigations.

Similarly, there is a need for an integrated bug-management approach. All code could be subject to static analysis first. Any code portion that cannot be proven bug-free should be subject to targeted fuzzing; any code that cannot adequately be reached by the fuzzer should be protected by specific run-time mitigations. Such an approach would contribute to minimizing exploitable vulnerabilities while preserving the performance of secure code.

BIBLIOGRAPHY

PUBLICATIONS PART OF THIS CUMULATIVE DISSERTATION

[1] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi. "Selfrando: Securing the Tor Browser Against De-anonymization Exploits." In: *Privacy Enhancing Technologies Symposium*. 2016. CORE2021 rank: A, CORE2014 rank: B. Included in Appendix A on Page 45.

[2] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi. "JITGuard: Hardening Just-in-time Compilers with SGX." In: *ACM Conference on Computer and Communications Security (CCS)*. 2017. CORE2021 rank: A*. Included in Appendix B on Page 61.

[3] F. Brasser, T. Frassetto, K. Riedhammer, A.-R. Sadeghi, T. Schneider, and C. Weinert. "VoiceGuard: Secure and Private Speech Processing." In: *Interspeech*. 2018. CORE2021 rank: A. Included in Appendix C on Page 77.

[4] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi. "DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization." In: *Annual Computer Security Applications Conference (ACSAC)*. 2019. CORE2021 rank: national USA, CORE2018 rank: A. Included in Appendix D on Page 83.

[5] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi. "FastKitten: Practical Smart Contracts on Bitcoin." In: *28th USENIX Security Symposium*. 2019. CORE2021 rank: A*. Included in Appendix E on Page 97.

[6] G. Dessouky, T. Frassetto, and A.-R. Sadeghi. "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments." In: *29th USENIX Security Symposium*. 2020. CORE2021 rank: A*. Included in Appendix F on Page 115.

[7] T. Frassetto, P. Jauernig, D. Koisser, and A.-R. Sadeghi. "CFInsight: A Comprehensive Metric for CFI Policies." In: *Network and Distributed System Security Symposium (NDSS)*. 2022. CORE2021 rank: A*. Included in Appendix G on Page 133.

OTHER PUBLICATIONS BY THE AUTHOR

[8] T. Frassetto. "Self-Rando: Practical Load-time Randomization Against Runtime Exploits." M.Sc. Thesis. Università degli Studi di Padova, 2016.

[9] H. Fereidooni, T. Frassetto, M. Miettinen, A.-R. Sadeghi, and M. Conti. "Fitness Trackers: Fit for Health but Unfit for Security and Privacy." In: *IEEE International Workshop on Safe, Energy-Aware, & Reliable Connected Health (CHASE-SEARCH)*. 2017.

[10] M. Miettinen, S. Marchal, I. Hafeez, T. Frassetto, N. Asokan, A.-R. Sadeghi, and S. Tarkoma. "IoT Sentinel Demo: Automated Device-Type Identification for Security Enforcement in IoT." In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2017.

[11] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX." In: *27th USENIX Security Symposium*. 2018.

[12] F. Brasser, L. Davi, A. Dhavlle, T. Frassetto, S. M. P. Dinakarrao, S. Rafatirad, A.-R. Sadeghi, A. Sasan, H. Sayadi, S. Zeitouni, and H. Homayoun. "Advances and Throwbacks in Hardware-assisted Security: Special Session." In: *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 2018.

[13] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi. "IMIX: In-Process Memory Isolation EXtension." In: *27th USENIX Security Symposium*. 2018.

[14] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. "Nautilus: Fishing for Deep Bugs with Grammars." In: *Network and Distributed System Security Symposium (NDSS)*. 2019.

[15] S. P. Bayerl, F. Brasser, C. Busch, T. Frassetto, P. Jauernig, J. Kolberg, A. Nautsch, K. Riedhammer, A.-R. Sadeghi, T. Schneider, E. Stapf, A. Treiber, and C. Weinert. "Poster: Privacy-preserving Speech Processing Via STPC and TEEs." In: *2nd Privacy Preserving Machine Learning (PPML) Workshop*. 2019.

[16] S. P. Bayerl, T. Frassetto, P. Jauernig, K. Riedhammer, A.-R. Sadeghi, T. Schneider, E. Stapf, and C. Weinert. "Offline Model Guard: Secure and Private ML on Mobile Devices." In: *23rd Design, Automation and Test in Europe Conference (DATE)*. 2020.

[17] J. Buchmann, G. Dessouky, T. Frassetto, Á. Kiss, A.-R. Sadeghi, T. Schneider, G. Traverso, and S. Zeitouni. "SAFE: A Secure and Efficient Long-Term Distributed Storage System." In: *8th International Workshop on Security in Blockchain and Cloud Computing (SBC)*. 2020.

[18] G. Dessouky, T. Frassetto, P. Jauernig, and A.-R. Sadeghi. "With Great Complexity Comes Great Vulnerability: Challenges of Secure Processor Design." In: *IEEE Security & Privacy* (2020).

[19] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi. "VoLTpwn: Attacking x86 Processor Integrity from Software." In: *29th USENIX Security Symposium*. 2020.

OTHER REFERENCES

[20] O. Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs." In: *Annual ACM Symposium on Theory of Computing*. 1987.

[21] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. "Efficient Software-Based Fault Isolation." In: *14th ACM Symposium on Operating Systems Principles*. 1993.

[22] O. Goldreich and R. Ostrovsky. "Software Protection and Simulation on Oblivious RAMs." In: *Journal of the ACM* (1996).

[23] PaX Team. *PaX Address Space Layout Randomization (ASLR)*. 2003. URL: http://pax.grsecurity.net/docs/aslr.txt.

[24] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. "Control-Flow Integrity." In: *ACM Conference on Computer and Communications Security (CCS)*. 2005.

[25] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. "CFI: Principles, Implementations, and Applications." In: *ACM Conference on Computer and Communications Security (CCS)*. 2005.

[26] S. Bhatkar, R. Sekar, and D. C. DuVarney. "Efficient Techniques for Comprehensive Protection from Memory Error Exploits." In: *14th USENIX Security Symposium*. 2005.

[27] M. Castro, M. Costa, and T. Harris. "Securing Software By Enforcing Data-flow Integrity." In: *7th USENIX Symposium on Operating Systems Design and Implementation*. 2006.

[28] S. McCamant and G. Morrisett. "Evaluating SFI for a CISC architecture." In: *15th USENIX Security Symposium*. 2006.

[29] H. Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)." In: *ACM Conference on Computer and Communications Security (CCS)*. 2007.

[30] Z. Wang and R. B. Lee. "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks." In: *Annual International Symposium on Computer Architecture*. 2007.

[31] D. Blazakis. "Interpreter Exploitation: Pointer Inference and JIT Spraying." In: *BlackHat DC*. 2010.

[32] W. De Groef, N. Nikiforakis, Y. Younan, and F. Piessens. "Jitsec: Just-in-time Security for Code Injection Attacks." In: *Benelux Workshop on Information and System Security (WISSEC)*. 2010.

[33] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. "Adapting Software Fault Isolation to Contemporary CPU Architectures." In: *18th USENIX Security Symposium*. 2010.

[34] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. "Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code." In: *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2011.

[35] P. Chen, Y. Fang, B. Mao, and L. Xie. "JITDefender: A defense against JIT spraying attacks." In: *IFIP International Information Security Conference*. 2011.

[36] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization." In: *21st USENIX Security Symposium*. 2012.

[37] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. "Privacy-preserving Group Data Access Via Stateless Oblivious RAM Simulation." In: *Annual ACM-SIAM symposium on Discrete Algorithms*. 2012.

[38] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. "ILR: Where'd My Gadgets Go?" In: *IEEE Symposium on Security and Privacy*. 2012.

[39] T. Kim, M. Peinado, and G. Mainar-Ruiz. "StealthMem: System-level Protection Against Cache-based Side Channel Attacks in the Cloud." In: *21st USENIX Security Symposium*. 2012.

[40] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code." In: *ACM Conference on Computer and Communications Security (CCS)*. 2012.

[41] P. Williams and R. Sion. "Round-optimal Access Privacy on Outsourced Storage." In: *ACM Conference on Computer and Communications Security (CCS)*. 2012.

[42] P. Chen, R. Wu, and B. Mao. "JITSafe: A Framework Against Just-in-time Spraying Attacks." In: *IET Information Security* (2013).

[43] L. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. "Gadge Me If You Can - Secure and Efficient Ad-hoc Instruction-Level Randomization for x86 and ARM." In: *ACM Symposium on Information, Computer and Communications Security*. 2013.

[44] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino. "Marlin: A Fine Grained Randomization Approach to Defend against ROP Attacks." In: *International Conference on Network and System Security*. 2013.

[45] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. "Librando: Transparent Code Randomization For Just-in-time Compilers." In: *ACM Conference on Computer and Communications Security (CCS)*. 2013.

[46] C. Liu, M. Hicks, and E. Shi. "Memory Trace Oblivious Program Execution." In: *IEEE Computer Security Foundations Symposium*. 2013.

[47] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. "Phantom: Practical Oblivious Computation in a Secure Processor." In: *ACM Conference on Computer and Communications Security (CCS)*. 2013.

[48] M. A. Pathak, B. Raj, S. Rane, and P. Smaragdis. "Privacy-Preserving Speech Processing: Cryptographic and String-Matching Frameworks Show Promise." In: *IEEE Signal Processing Magazine* (2013).

[49] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi. "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization." In: *IEEE Symposium on Security and Privacy*. 2013.

[50] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. "Path ORAM: An Extremely Simple Oblivious RAM Protocol." In: *ACM Conference on Computer and Communications Security (CCS)*. 2013.

[51] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. "Practical Control Flow Integrity and Randomization for Binary Executables." In: *IEEE Symposium on Security and Privacy*. 2013.

[52] M. Zhang and R. Sekar. "Control Flow Integrity for COTS Binaries." In: *22nd USENIX Security Symposium*. 2013.

[53] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. "You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code." In: *ACM Conference on Computer and Communications Security (CCS)*. 2014.

[54] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. "Hacking Blind." In: *IEEE Symposium on Security and Privacy*. 2014.

[55] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection." In: *23rd USENIX Security Symposium*. 2014.

[56] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. "Out of Control: Overcoming Control-flow Integrity." In: *IEEE Symposium on Security and Privacy*. 2014.

[57] B. Niu and G. Tan. "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity." In: *ACM Conference on Computer and Communications Security (CCS)*. 2014.

[58] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. "Enforcing Forward-edge Control-flow Integrity in GCC & LLVM." In: *23rd USENIX Security Symposium*. 2014.

[59] ARM. *TrustZone*. URL: https://developer.arm.com/ip-products/security-ip/trustzone.

[60] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis. "The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines." In: *Network and Distributed System Security Symposium (NDSS)*. 2015.

[61] J. Corbet. *Memory Protection Keys*. 2015. URL: https://lwn.net/Articles/643797/.

[62] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity." In: *Network and Distributed System Security Symposium (NDSS)*. 2015.

[63] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. "Readactor: Practical Code Randomization Resilient to Memory Disclosure." In: *36th IEEE Symposium on Security and Privacy*. 2015.

[64] J. Gionta, W. Enck, and P. Ning. "HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities." In: *5th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2015.

[65] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz. "Large-scale Automated Software Diversity—Program Evolution Redux." In: *IEEE Transactions on Dependable and Secure Computing* (2015).

[66] P. Koeberl, V. Phegade, A. Rajan, T. Schneider, S. Schulz, and M. Zhdanova. "Time to Rethink: Trust Brokerage Using Trusted Execution Environments." In: *Trust and Trustworthy Computing (TRUST)*. 2015.

[67] G. Koppen. *Include SelfRando Patches Into Our Hardened Builds*. 2015. URL: https://gitlab.torproject.org/legacy/trac/-/issues/17406.

[68] R. Kumaresan, T. Moran, and I. Bentov. "How To Use Bitcoin To Play Decentralized Poker." In: *ACM Conference on Computer and Communications Security (CCS)*. 2015.

[69] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. "Ghostrider: a Hardware-Software System for Memory Trace Oblivious Computation." In: *ACM SIGARCH Computer Architecture News* (2015).

[70] J. de Mooij. *W^X JIT-code Enabled in Firefox*. 2015. URL: https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox.

[71] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. "Observing and Preventing Leakage in MapReduce." In: *ACM Conference on Computer and Communications Security (CCS)*. 2015.

[72] PaX Team. *RAP: RIP ROP*. 2015. URL: https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf.

[73] M. Payer, A. Barresi, and T. R. Gross. "Fine-grained Control-flow Integrity Through Binary Hardening." In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2015.

[74] A. Rane, C. Lin, and M. Tiwari. "Raccoon: Closing Digital Side-channels Through Obfuscated Execution." In: *24th USENIX Security Symposium*. 2015.

[75] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas. "Constants Count: Practical Improvements to Oblivious RAM." In: *24thUSENIX Security Symposium*. 2015.

[76] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. "Exploiting and Protecting Dynamic Code Generation." In: *Network and Distributed System Security Symposium (NDSS)*. 2015.

[77] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. "Practical Context-Sensitive CFI." In: *ACM Conference on Computer and Communications Security (CCS)*. 2015.

[78] Y. Xu, W. Cui, and M. Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems." In: *IEEE Symposium on Security and Privacy*. 2015.

[79] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. "Leakage-Resilient Layout Randomization for Mobile Devices." In: *Network and Distributed System Security Symposium (NDSS)*. 2016.

[80] V. Costan and S. Devadas. *Intel SGX Explained*. Tech. rep. Cryptology ePrint Archive, 2016. URL: https://eprint.iacr.org/2016/086.pdf.

[81] V. Costan, I. A. Lebedev, and S. Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation." In: *26th USENIX Security Symposium*. 2016.

[82] X. Ge, N. Talele, M. Payer, and T. Jaeger. "Fine-Grained Control-Flow Integrity for Kernel Software." In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2016.

[83] G. Gong. *Pwn a Nexus Device With a Single Vulnerability*. 2016. URL: https://web.archive.org/web/20210307205130/https://cansecwest.com/slides/2016/CSW2016%5C_Gong%5C_Pwn%5C_a%5C_Nexus%5C_device%5C_with%5C_a%5C_single%5C_vulnerability.pdf.

[84] M. Holman. *Out-of-process JIT Support*. 2016. URL: https://github.com/Microsoft/ChakraCore/pull/1561.

[85] Intel. *Software Guard Extensions (SGX)*. URL: https://www.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[86] K. A. Küçük, A. Paverd, A. Martin, N. Asokan, A. Simpson, and R. Ankele. "Exploring the Use of Intel SGX for Secure Many-Party Applications." In: *System Software for Trusted Execution (SysTEX)*. 2016.

[87] R. Kumaresan and I. Bentov. "Amortizing Secure Computation with Penalties." In: *ACM Conference on Computer and Communications Security (CCS)*. 2016.

[88] R. Kumaresan, V. Vaikuntanathan, and P. N. Vasudevan. "Improvements to Secure Computation with Penalties." In: *ACM Conference on Computer and Communications Security (CCS)*. 2016.

[89] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. "ARMageddon: Cache attacks on mobile devices." In: *25th USENIX Security Symposium*. 2016.

[90] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. "CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing." In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016.

[91] F. Liu, H. Wu, K. Mai, and R. B. Lee. "Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks." In: *International Symposium on Microarchitecture (MICRO)*. 2016.

[92] O. Ohrimenko, F. Schuster, C. Fournet, A. Meht, S. Nowozin, K. Vaswani, and M. Costa. "Oblivious Multi-Party Machine Learning on Trusted Processors." In: *25th USENIX Security Symposium*. 2016.

[93] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. "A Tough Call: Mitigating Advanced Code-reuse Attacks at the Binary Level." In: *IEEE Symposium on Security and Privacy*. 2016.

[94] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh. "SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection." In: *Annual Design Automation Conference (DAC)*. 2016.

[95] Z. Zhou, M. K. Reiter, and Y. Zhang. "A Software Approach to Defeating Side Channels in Last-Level Caches." In: *ACM Conference on Computer and Communications Security (CCS)*. 2016.

[96] AMD. *Secure Encrypted Virtualization (SEV)*. URL: https://developer.amd.com/sev/.

[97] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi. "Secure Multiparty Computation from SGX." In: *Financial Cryptography and Data Security (FC)*. 2017.

[98] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical." In: *11th USENIX Workshop on Offensive Technologies (WOOT)*. 2017.

[99] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. "Control-flow Integrity: Precision, Security, and Performance." In: *ACM Comput. Surv.* (2017).

[100] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu." In: *ACM Asia Conference on Computer and Communications Security (ASIACCS)*. 2017.

[101] G. Dessouky, E. Stapf, P. Mahmoody, A. Gruler, and A.-R. Sadeghi. "Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures." In: *ACM Conference on Computer and Communications Security (CCS)*. 2017.

[102] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee. "Efficient Protection of Path-Sensitive Control Security." In: *26th USENIX Security Symposium*. 2017.

[103] M. Elsabagh, D. Fleck, and A. Stavrou. "Strict Virtual Call Integrity Checking for C++ Binaries." In: *ACM Asia Conference on Computer and Communications Security*. 2017.

[104] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi. "HardIDX: Practical and Secure Index with SGX." In: *Conference on Data and Applications Security and Privacy (DBSec)*. 2017.

[105] C. Glackin, G. Chollet, N. Dugan, N. Cannings, J. Wall, S. Tahir, I. G. Ray, and M. Rajarajan. "Privacy Preserving Encrypted Phonetic Search of Speech Data." In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2017.

[106] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. "Cache Attacks on Intel SGX." In: *European Workshop on Systems Security*. 2017.

[107] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory." In: *26th USENIX Security Symposium*. 2017.

[108] J. Liu, M. Juuti, Y. Lu, and N. Asokan. "Oblivious Neural Network Predictions via MiniONN Transformations." In: *ACM Conference on Computer and Communications Security (CCS)*. 2017.

[109] G. Maisuradze, M. Backes, and C. Rossow. "Dachshund: Digging for and Securing Against (non-) Blinded Constants in JIT Code." In: *Network and Distributed System Security Symposium (NDSS)*. 2017.

[110] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry. *Sprites: Payment Channels that Go Faster than Lightning*. Tech. rep. arXiv:1702.05812, 2017.

[111] M. Miller. *Mitigating Arbitrary Native Code Execution in Microsoft Edge*. 2017. URL: https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/.

[112] A. Moghimi, G. Irazoqui, and T. Eisenbarth. "CacheZoom: How SGX Amplifies The Power of Cache Attacks." In: *International Conference on Cryptographic Hardware and Embedded Systems*. 2017.

[113] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida. "MARX: Uncovering Class Hierarchies in C++ Programs." In: *Network and Distributed System Security Symposium (NDSS)*. 2017.

[114] J. Poon and V. Buterin. *Plasma: Scalable Autonomous Smart Contracts*. White paper. 2017.

[115] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2017.

[116] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs." In: *Network and Distributed System Security Symposium (NDSS)*. 2017.

[117] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs." In: *Network and Distributed System Security Symposium (NDSS)*. 2017.

[118] R. Sinha, S. Rajamani, and S. A. Seshia. "A Compiler and Verifier for Page Access Oblivious Computation." In: *11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE*. 2017.

[119] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution." In: *26th USENIX Security Symposium*. 2017.

[120] P. Alcorn. *Intel Unveils Cascade Lake, In-Silicon Spectre and Meltdown Mitigations*. 2018. URL: https://www.tomshardware.com/news/intel-cascade-lake-details-spectre-meltdown,37674.html.

[121] S. Cui, S. Belguith, M. Zhang, M. R. Asghar, and G. Russello. "Preserving Access Pattern Privacy in SGX-Assisted Encrypted Search." In: *27th International Conference on Computer Communication and Networks (ICCCN)*. 2018.

[122] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi. "On the Effectiveness of Type-based Control Flow Integrity." In: *Annual Computer Security Applications Conference (ACSAC)*. 2018.

[123] R. Gawlik and T. Holz. "SoK: Make JIT-spray Great Again." In: *12th USENIX Workshop on Offensive Technologies (WOOT)*. 2018.

[124] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks." In: *27th USENIX Security Symposium*. 2018.

[125] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee. "Enforcing Unique Code Target Property for Control-Flow Integrity." In: *ACM Conference on Computer and Communications Security (CCS)*. 2018.

[126] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. "GAZELLE: A Low Latency Framework for Secure Neural Network Inference." In: *27th USENIX Security Symposium*. 2018.

[127] H. A. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. "Arbitrum: Scalable, Private Smart Contracts." In: *27th USENIX Security Symposium*. 2018.

[128] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. "Compiler-Assisted Code Randomization." In: *39th IEEE Symposium on Security and Privacy*. 2018.

[129] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert. "τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries." In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. 2018.

[130] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks." In: *USENIX Annual Technical Conference (ATC)*. 2018.

[131] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. "Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications." In: *ACM Asia Conference on Computer and Communications Security (ASIACCS)*. 2018.

[132] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. "DeepSecure: Scalable Provably-Secure Deep Learning." In: *Annual Design Automation Conference (DAC)*. 2018.

[133] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla. "Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems." In: *Annual Design Automation Conference (DAC)*. 2018.

[134] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *27th USENIX Security Symposium*. 2018.

[135] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee. "Obfuscuro: A Commodity Obfuscation Engine on Intel SGX." In: *Network and Distributed System Security Symposium (NDSS)*. 2019.

[136] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts." In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019.

[137] Intel. *INTEL-SA-00233: Microarchitectural Data Sampling Advisory*. 2019. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html.

[138] M. Miller. *Trends, Challences, and Stratecic Shifts in the Software Vulnerability Mitication Landscape*. BlueHat IL. 2019. URL: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[139] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert. "Analyzing Control Flow Integrity with LLVM-CFI." In: *Annual Computer Security Applications Conference (ACSAC)*. 2019.

[140] M. Orenbach, Y. Michalevsky, C. Fetzer, and M. Silberstein. "CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves." In: *USENIX Annual Technical Conference (ATC)*. 2019.

[141] M. J. Reed. *New Confidential Computing Solutions Emerge on the Hyperledger Avalon Trusted Compute Framework*. 2019. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/new-confidential-computing-solutions-emerge-on-the-hyperledger-avalon-trusted-compute.html.

[142] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization." In: *28th USENIX Security Symposium*. 2019.

[143] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. "Plundervolt: Software-based Fault Injection Attacks Against Intel SGX." In: *IEEE Symposium on Security and Privacy*. 2020.

[144] M. Orenbach, A. Baumann, and M. Silberstein. "Autarky: Closing Controlled Channels with Self-Paging Enclaves." In: *15th European Conference on Computer Systems*. 2020.

[145] T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz. "NoJITsu: Locking Down JavaScript Engines." In: *Network and Distributed System Security Symposium (NDSS)*. 2020.

[146] S. Priyadarshan, H. Nguyen, and R. Sekar. "Practical Fine-Grained Binary Code Randomization." In: *Annual Computer Security Applications Conference (ACSAC)*. 2020.

[147] MITRE Corporation. *2021 CWE Top 25 Most Dangerous Software Weaknesses*. 2021. URL: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[148] G. Saileshwar and M. Qureshi. "MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design." In: *30th USENIX Security Symposium*. 2021.

[149] S. Özkan. *Browse CVE Vulnerabilities By Date*. 2022. URL: https://www.cvedetails.com/browse-by-date.php.

LIST OF ACRONYMS

AIR Average Indirect-target Reduction
ASLR Address Space Layout Randomization
CAT (Intel) Cache Allocation Technology
CFG Control-Flow Graph
CFI Control-Flow Integrity
DFI Data-Flow Integrity
IPC Inter-Process Communication
JIT Just-In-Time
LLC Last-Level Cache
MPC Secure Multi-Party Computation
MPK (Intel) Memory Protection Keys
ORAM Oblivious RAM
ROP Return-Oriented Programming
RAM Random-Access Memory
SEV (AMD) Secure Encrypted Virtualization
SFI Software Fault Isolation
SGX (Intel) Software Guard Extensions
SMT Simultaneous Multi-Threading
TDX (Intel) Trust Domain Extensions
TEE Trusted Execution Environment
TSX (Intel) Transactional Synchronization Extensions
TZ (ARM) TrustZone

ERKLÄRUNG GEMÄß §9 DER PROMOTIONSORDNUNG

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, März 2022

Tommaso Frassetto

Part II

PUBLICATIONS PART OF THIS CUMULATIVE DISSERTATION

This part contains a copy of the proceedings version of every publication that composes this cumulative dissertation.

The paper *"Selfrando: Securing the Tor Browser Against De-anonymization Exploits"* [1], which was authored by M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, and was published at PETS 2016 (core2021 rank: a, core2014 rank: b), starts on page 45.

The paper *"JITGuard: Hardening Just-in-time Compilers with SGX"* [2], which was authored by T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi, and was published at ACM CCS 2017 (core2021 rank: a*), starts on page 61.

The paper *"VoiceGuard: Secure and Private Speech Processing"* [3], which was authored by F. Brasser, T. Frassetto, K. Riedhammer, A.-R. Sadeghi, T. Schneider, and C. Weinert, and was published at Interspeech 2018 (core2021 rank: a), starts on page 77.

The paper *"DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization"* [4], which was authored by F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, and was published at ACSAC 2019 (core2021 rank: national usa, core2018 rank: a), starts on page 83.

The paper *"FastKitten: Practical Smart Contracts on Bitcoin"* [5], which was authored by P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, and was published at USENIX Security 2019 (core2021 rank: a*), starts on page 97.

The paper *"HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments"* [6], which was authored by G. Dessouky, T. Frassetto, and A.-R. Sadeghi, and was published at USENIX Security 2020 (core2021 rank: a*), starts on page 115.

The paper *"CFInsight: A Comprehensive Metric for CFI Policies"* [7], which was authored by T. Frassetto, P. Jauernig, D. Koisser, and A.-R. Sadeghi, and was published at NDSS 2022 (core2021 rank: a*), starts on page 133.

Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi

Selfrando: Securing the Tor Browser against De-anonymization Exploits

Abstract: Tor is a well-known anonymous communication system used by millions of users, including journalists and civil rights activists all over the world. The Tor Browser gives non-technical users an easy way to access the Tor Network. However, many government organizations are actively trying to compromise Tor not only in regions with repressive regimes but also in the free world, as the recent FBI incidents clearly demonstrate. Exploiting software vulnerabilities in general, and browser vulnerabilities in particular, constitutes a clear and present threat to the Tor software. The Tor Browser shares a large part of its attack surface with the Firefox browser. Therefore, Firefox vulnerabilities (even patched ones) are highly valuable to attackers trying to monitor users of the Tor Browser.

In this paper, we present selfrando—an enhanced and practical load-time randomization technique for the Tor Browser that defends against exploits, such as the one FBI allegedly used against Tor users. Our solution significantly improves security over standard address space layout randomization (ASLR) techniques currently used by Firefox and other mainstream browsers. Moreover, we collaborated closely with the Tor Project to ensure that selfrando is fully compatible with AddressSanitizer (ASan), a compiler feature to detect memory corruption. ASan is used in a hardened version of Tor Browser for test purposes. The Tor Project decided to include our solution in the hardened releases of the Tor Browser, which is currently undergoing field testing.

Keywords: De-anonymization exploits, code-randomization, privacy-oriented software, Tor Browser.

DOI 10.1515/popets-2016-0050
Received 2016-02-29; revised 2016-06-02; accepted 2016-06-02.

1 Introduction

The Tor Project provides a suite of free software and a worldwide network designed to facilitate anonymous information exchange and to prevent surveillance and fingerprinting of these interactions. The Tor network is open to anyone and widely used by civil rights activists, whistleblowers, journalists, citizens of oppressive regimes, etc. Many sensitive websites, including the late Silk Road black market, are only accessible over Tor. Consequently, the Tor Network is continually facing de-anonymization attacks by law enforcement, intelligence agencies, and foreign nation states. A de-anonymization attack aims to disclose information, such as the identity or the location, of an anonymous user. While many de-anonymization attacks rely on weaknesses in the network protocol, they often require that adversaries control a large number of Tor nodes [26] or only work in a lab environment [39].

An alternative and practical way to de-anonymize Tor users is to exploit security vulnerabilities in the software used to access the Tor network. The most common way to access Tor is via the Tor Browser (TB) [73], which includes a pre-configured Tor client. Since TB is based on Mozilla's Firefox browser, they share a large part of their attack surfaces. In 2013, the Federal Bureau of Investigation (FBI) exploited a known software vulnerability in Firefox [71] to de-anonymize Tor users that had not updated to the most recent version of TB [27, 57, 74]. Due to the success of this operation, exploit brokers [79] (and, presumably, governments and criminals) are currently soliciting exploits for the TB. In early 2016, it was confirmed that the FBI continues to monitor the Tor network, this time using a de-

Mauro Conti: Università degli Studi di Padova, E-mail: conti@math.unipd.it
Stephen Crane: Immunant, Inc., E-mail: sjc@immunant.com
Tommaso Frassetto: CASED/Technische Universität Darmstadt, Germany, E-mail: tommaso.frassetto@trust.cased.de
Andrei Homescu: Immunant, Inc., E-mail: ah@immunant.com
Georg Koppen: The Tor Project, E-mail: gk@torproject.org
Per Larsen: Immunant, Inc., E-mail: perl@immunant.com
Christopher Liebchen: CASED/Technische Universität Darmstadt, Germany, E-mail: christopher.liebchen@trust.cased.de
Mike Perry: The Tor Project, E-mail: mikeperry@torproject.org
Ahmad-Reza Sadeghi: CASED/Technische Universität Darmstadt, Germany, E-mail: ahmad.sadeghi@trust.cased.de

anonymization attack devised by Carnegie Mellon University researchers [19].

The Open Technology Fund commissioned a study on current and future hardening efforts to reduce the attack surface of the TB [58]. One of the recommendations was to use compiler techniques to detect memory corruption (buffer overflow, use-after-free, uninitialized variables, etc.) such as the AddressSanitizer (ASan) feature [61]. Another key recommendation was to use address space layout randomization (ASLR) to prevent exploitation of memory corruption vulnerabilities. While ASan imposes a high runtime overhead [61], ASLR is very efficient. However, ASLR was recommended because it is widely supported by compilers and operating systems, not for its security properties. In fact, the shortcomings of ASLR are well documented in the academic literature [8, 16, 33, 62, 64, 68]. ASLR can be made significantly stronger by randomizing not just the base address of modules but also the code inside each module. Address space layout permutation (ASLP) [44], for instance, randomizes the location of each function individually, thwarting many of the techniques used to bypass ASLR. Until now, however, the ASLR improvements suggested in the literature have suffered from one or more drawbacks that have prevented their use in practice. Some techniques rely on binary rewriting, which does not scale to complex programs such as web browsers [22, 38]; others randomize the code using a customized compiler [35], or require each user to download their own unique binary [42].

Goals and Contributions The goal of this paper is to demonstrate a load-time randomization technique—named selfrando—that improves security over ASLR while preserving the features that enabled ASLR's widespread adoption. While technically challenging, our use of load-time function layout permutation ensures that the attack surface changes from one run to another. Load-time randomization also ensures compatibility with code signing and distribution mechanisms that use caching to efficiently serve millions of users. Finally, we worked in close collaboration with the TB developers to ensure that selfrando was fully compatible with ASan so that users can use both techniques simultaneously. ASan is used in a hardened version of TB to detect and diagnose memory corruption errors.

Summing up, our main contributions are:

- **Practical Randomization Framework** Unlike other solutions that have only been tested on benchmarks, selfrando can be applied to the TB without any changes to the source code. To the best of our knowledge, selfrando is the first approach that avoids risky binary rewriting or the need to use a custom compiler, and instead works with existing build tools. Moreover, it is fully compatible with ASan, which required additional implementation effort since the randomization interferes with ASan.
- **Increased Entropy and Leakage Resilience** selfrando reduces the impact of information leakage vulnerabilities and increases entropy relative to ASLR, making selfrando more effective against guessing attacks. Our use of load-time randomization mitigates threats from attackers observing binaries during download or after the executable files have been stored on disk.
- **Hardening the Tor Browser** We demonstrate the practicality of selfrando by applying it to the entire TB without requiring any code changes. Our detailed and careful evaluation shows that the startup and performance overheads of selfrando are negligible.

2 Background

2.1 Exploiting Memory Corruption

Unlike modern programming languages, C and C++ rely on manual memory management, trading reliability for flexibility and performance. Hence, memory management errors often create vulnerabilities that can be exploited to hijack control flow and perform other malicious operations that were never intended by the program authors.

Traditionally, attackers used a buffer overflow to directly inject malicious code into a program and execute it [6]. However, the introduction of the W⊕X policy that requires memory pages to either be writable or executable, but not both, made most code-injection attacks [49] obsolete. As W⊕X became commonplace, attackers changed their tactics from code injection to code reuse. These attacks reuse existing, legitimate code for malicious purposes and have therefore proven far harder to stop than code injection. Return-into-`libc` (RILC) attacks, for example, arrange the stack contents so the attacker can call dangerous functions inside the C library with attacker-controlled arguments [52]. Such attacks were later generalized to the so-called return-oriented programming (ROP) [63]. The insight behind ROP is that attackers can build a malicious virtual machine out of short instruction sequences—called *gadgets* in ROP parlance—ending with a return (or some other

indirect branch). These gadgets are all located inside application code, so the attacker has no need to inject them into the program. Over the last decade researchers have discovered many variants of code-reuse attacks [10, 11, 15, 59, 76], most of which are not stopped by ASLR, W⊕X, or other widely deployed exploit mitigations.

2.2 Preventing Code-Reuse Exploits

To successfully mount a code-reuse attack, several requirements must be met. First, the application must contain a memory corruption vulnerability that allows control flow to be hijacked. Techniques such as control-flow integrity and stack canaries make control-flow hijacking harder but do not prevent it outright [14, 29, 36, 48]. Another key requirement is knowledge of the absolute addresses of the gadgets to reuse. In principle, ASLR [54] prevents adversaries from knowing the absolute locations of ROP gadgets. However, since ASLR only randomizes the base address of a library, adversaries still know the relative positions of all functions inside a library. Using this knowledge together with a leaked code pointer, attackers can compute the absolute addresses of all functions in the same library. Academics have documented numerous ways to leak code or pointers to code [24, 60, 62, 66]. Permuting the functions inside a library removes attackers' knowledge of the relative function layout inside each library, and additionally improves entropy by allowing an exponentially higher number of code layouts in comparison to ASLR [44].

Numerous other fine-grained diversity techniques have been suggested in the literature. In this paper, we focus on function permutation since it is practical and efficient, as shown by existing diversity surveys [17, 46].

Unfortunately, previous fine-grained diversity approaches have been unable to replace ASLR because they have one or more of the following drawbacks:

1. they introduce unacceptable performance overheads [69],
2. they rely on unsafe binary rewriting techniques that do not scale to complex, real-world applications,
3. they randomize code at compile time which is incompatible with current distribution mechanisms that are optimized to deliver a single binary.

In contrast, the technique we present in this paper, selfrando, avoids all of these drawbacks and scales to real-world applications including Firefox and TB.

2.3 Trust in Privacy-preserving Software

As we have previously mentioned, any tactic that allows de-anonymization of Tor network users is likely to be attempted by law enforcement, intelligence agencies, and other resourceful adversaries. The ability to surreptitiously insert backdoors into the TB would be a particularly powerful attack. In order to reduce the likelihood that backdooring attempts would go unnoticed, the Tor developers ensure that builds are reproducible. Even though the TB source code can be downloaded by anyone, differences in build tools, libraries, file system layout and even system time make it hard to simply build the TB from sources and compare it to the official binaries to ensure the absence of backdoors. Therefore, the TB is built using Gitian, a special tool which provides a reproducible build environment [55, 72]. This allows third parties to independently compile and verify the binaries distributed by the Tor Project and detect any signs of external compromise.

Gitian consists of a virtual machine and a number of build scripts to automate the process. The virtual machine insulates the build from the outside environment. At the time selfrando was developed, the TB builds for Linux used a virtual machine based on Ubuntu 10.04. Hence, many build tools were unavailable or outdated. To cope with this shortcoming, we either compiled a recent version of the tool in the virtual machine itself or we adapted the build process to support the older version. The Tor developers recently (May 2016) switched to a virtual machine based on Debian 7. During the switch no modifications were necessary to our code.

3 Selfrando

3.1 Design Objectives

Our main objective is to substantially raise the costs for attackers to exploit memory-corruption vulnerabilities. For practicality reasons, we choose to support complex C/C++ programs (e.g., a browser) without modifying their source code. Further, we retain full compatibility with current build systems, i.e., we should avoid any modification to compilers, linkers, and other operating system components. To be applicable for privacy-preserving open-source tools, we must not rely on any third-party proprietary software. Finally, our solution should not substantially increase the size of the program in memory or on disk.

3.2 Threat Model

We make standard assumptions from underlying real-world adversary settings: we assume that a remote attacker triggers a memory corruption vulnerability to hijack control flow and achieve remote code execution. Due to the widespread deployment of stack protections (e.g., StackGuard [18] and SafeStack [70]) and the fact that most exploits against browsers rely on use-after-free errors [75], we assume that the adversary exploits a heap-based memory corruption vulnerability. This means that the adversary can use code pointers stored on the heap to disclose the location of code before mounting a code-reuse attack. Further, we assume that a W⊕X policy is in place to prevent code injection, which is true for all modern systems. In this work we do not consider attacks that target the browser's JIT engine.

Note that our threat model does not cover some theoretical attacks such as JIT-ROP [66] and COOP [59] that have only been demonstrated in an academic setting. As mentioned above, our main objective is high practicality while significantly improving security provided by ASLR against memory corruption attacks; defenses that can stop JIT-ROP and COOP are less efficient and rely on special hardware support, a custom compiler, and a patched OS kernel [12, 20, 21].

3.3 Selfrando Design

Existing exploit mitigations such as W⊕X and ASLR already make de-anonymization exploits costly to develop. Thus, exploits which bypass these mitigations often target high-profile applications with many users. Although the Tor user base isn't large, the TB shares a large amount of code with Firefox which has hundreds of millions of users and contains more than 20 million lines of code. The similarities between the TB and Firefox make it comparatively easy to re-purpose mainstream Firefox exploits to de-anonymize Tor users. We can use our improved randomization mechanism to protect the TB and at the same time strongly raise the bar for the adversary to port exploits from Firefox to TB.

The easiest way to perform fine-grained code randomization is by customizing the compiler to take a seed value and generate a randomized binary [32, 42]. Unfortunately, compiling and distributing a unique binary for each is impractical for introducing diversity among a population of programs [30, 78]. With more implementation effort, we can delay randomization until load-time,

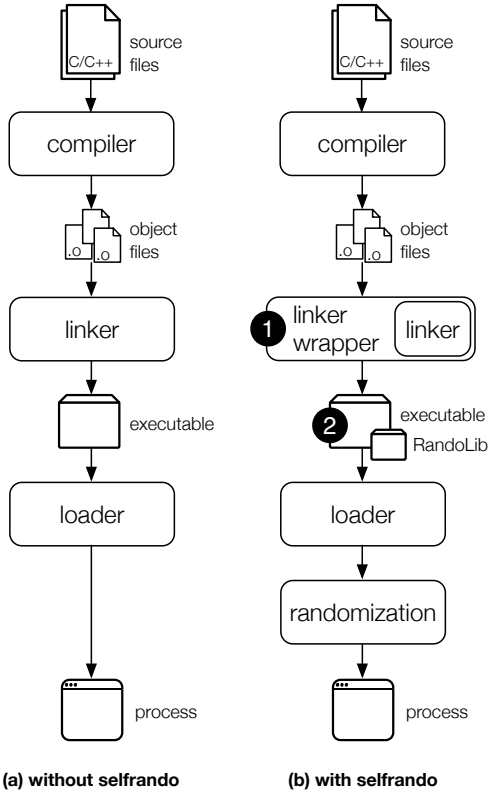

Fig. 1. Building and running applications without (a) and with selfrando (b) enabled.

which has several benefits. Most importantly, software vendors only need to compile and test a single binary. A single binary also means that users can continue to use hashes to verify the authenticity of the downloaded binary. Finally, modern content delivery networks rely extensively on caching binaries on servers; this optimization is no longer possible with unique binaries.

In the context of privacy-preserving software such as TB, compile-time randomization raises additional challenges. Randomized builds would complicate the deterministic build process,[1] which is important to increase trust in the distributed binary (see Section 2.3). Moreover, compile-time randomization would (a) increase the feasibility of a de-anonymization attack due to individual, observable characteristics of a particular build, and (b) allow an attacker to build knowledge of the mem-

1 A randomized build can be implemented in a deterministic environment by passing a random seed as an input to the deterministic process. The builds would then be distributed along with their seed. A user could then check the integrity of her build by running the deterministic process again with the same seed. However, that check would not prove the integrity of builds with other seeds.

ory layout across application restarts, since the layout would be fixed.

For these reasons, we decided to develop a framework which makes the program binary randomize itself at load time. We chose function permutation (ASLP) as the randomization granularity, since it dramatically increases[2] the entropy of the code layout while imposing the same low overheads as ASLR [44]. Since discovering function boundaries at load-time by analyzing the program binary is unreliable and does not scale to large programs, we pre-compute these boundaries statically and store the necessary information in each binary. We call this *Translation and Protection* (TRaP) information.

Rather than modifying the compiler or linker, we developed a small tool which wraps the system linker, extracts all function boundaries from the object files used to build the binary, then appends the necessary TRaP information to the binary itself. Our linker wrapper works with the standard compiler toolchains on Linux and Windows and only requires a few changes to the build scripts to use with the TB.

Figure 1a represents the usual workflow from the C/C++ source code to a running program. Figure 1b represents the modified workflow with selfrando. A *linker wrapper* intercepts calls to the linker and calls selfrando to gather information on the executable file ❶. Then, it embeds TRaP information and a load-time randomization library, RandoLib, into the binary file ❷. When the loader loads the application, it will invoke RandoLib instead of the entry point of the application. RandoLib will randomize the order of the functions in memory and then transfer control to the original program entry point.

4 Implementation

One of our main goals is to demonstrate the practicality of selfrando by integrating it into the TB. To test selfrando before it is released to Tor users at large, the Tor project decided to first include our defense in a series of experimental, hardened builds for Linux.[3] The hardened builds of Tor include additional features such as AddressSanitizer (ASan), a compiler feature which can detect memory corruption. ASan and selfrando are complementary in nature. The former detects bugs that can create security issues, however, ASan is not a defense mechanism like selfrando and should not be relied upon to stop exploits [51].

To build a program with selfrando, the build scripts must be updated to use our linker wrapper rather than directly invoking the system linker. The wrapper accepts the same arguments as the system linker, so modifying the build scripts is a straightforward task for a skilled software developer. This enables us to intercept any invocation of the linker and modify its arguments. In the following we will explain the major implementation aspects with the help of Figure 2. Notably, we will explain in detail how selfrando (1) extracts the metadata needed to create self-randomizing binaries, (2) embeds the extracted information and the load-time component into the generated binary, and (3) permutes all functions during load time without breaking the application.

Finally, we describe two practical challenges that we solved to make selfrando compatible with the hardened build of TB. Specifically, we needed selfrando to (4) support stack unwinding which is needed for stack traces and exception handling and (5) be compatible with ASan.

4.1 Extracting TRaP Information

When a module is loaded, selfrando permutes the order of all its functions. To do so, selfrando requires accurate information about function boundaries. If this information is not accurate, shuffling the function layout may inadvertently introduce errors that prevent correct execution of the application. After a function is moved, all references and pointers to this function, e.g., the target address of a call, become invalid because they still reference the old address. Hence, selfrando needs to update all references to the moved function, and therefore requires, for each function, a complete list of all locations that reference that function.

Such information is present in the intermediate object files ❶. Since this metadata is usually not required during execution, the linker strips it from the final binary. Our linker wrapper therefore intercepts the linking process to extract function boundaries and references and embeds this information for use at load time.

However, object files do not explicitly mark all function references. Specifically, we found that in some cases the compiler optimizes the code by inserting direct jumps between two functions. Such references are not

2 We compare the entropy of function permutation and ASLR in Section 5.

3 Selfrando is also compatible with Android and closed-source platforms such as Microsoft Windows.

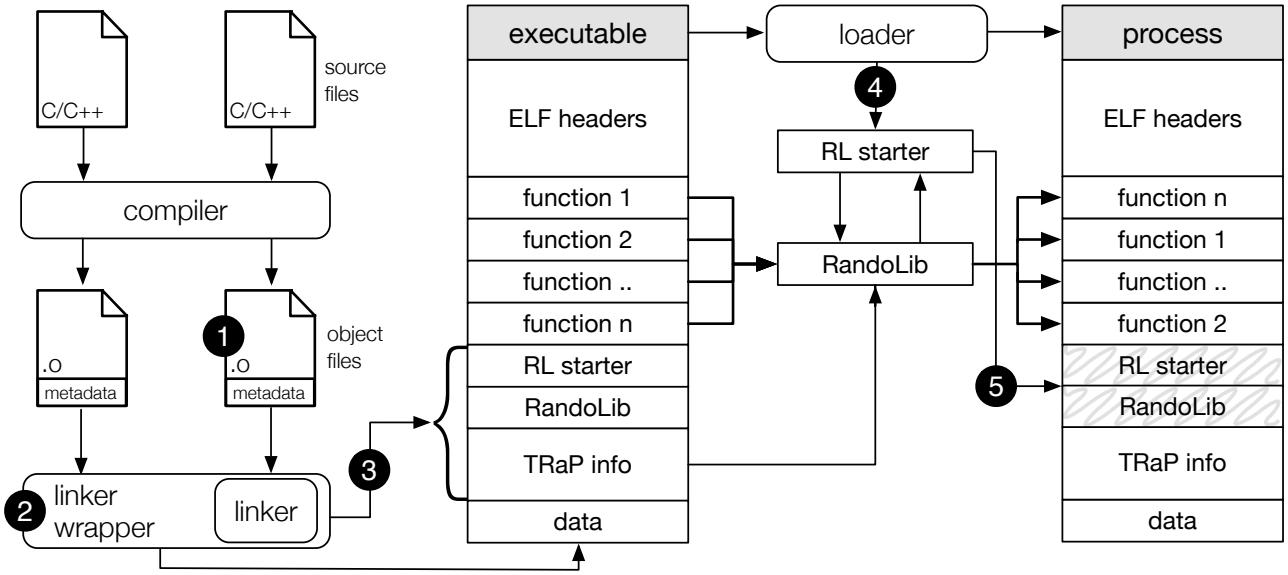

Fig. 2. Workflow of selfrando.

marked with an explicit relocation because they are already resolved by the compiler. Fortunately, we can disable this behavior with a compiler option causing the compiler to place each function in a separate section. Since the compiler marks all references between sections, we can then see all function references. While enabling this option slightly increases build-time (0.07%), it also enables a linker optimization which increases locality [31].

Pre-compiled language runtime object files are another obstacle. One example is `crtbegin.o` for GCC which contains functions to initialize the runtime environment for applications that were programmed in C. In our current implementation, we treat such object files as one single block because they contain only a few functions. This has a negligible impact on the overall randomization entropy. Nevertheless, we are currently investigating how we can generate selfrando-compatible versions of the pre-compiled object files.

After selfrando extracts the necessary metadata from each generated object, it adds an additional linker argument that instructs the linker to generate a *map file*, which is a text file that contains the memory layout of the final binary ❷. Using the metadata and the map file, selfrando can compute the final location of each function in the executable file and all references to these functions.

Next, we explain how we embed the TRaP information in the binary to make it available to the run-time component—RandoLib.

4.2 Embedding TRaP information

We include the TRaP info, which is used by RandoLib, in the executable to make selfrando self-contained. This avoids having to manage additional files, which could add logistical burden.

However, from a technical point of view, embedding the data in a space-efficient and binary format-compatible way without modifying the linker is challenging. The main reasons are that (1) some of the metadata is only available *after* linking is complete, and (2) we cannot pre-allocate space for the data since the exact amount of space needed is unknown until linking is done. In particular, the start address of each function in the linked binary is determined by the order and final addresses of the object files in the binary, and therefore unknown until all objects are linked.

To add additional data to the final binary, we have to resort to a trick that involves changing the linker input so that it adds an empty segment header in the beginning of the binary. Note that a linked ELF binary is divided into segments. The linker creates a segment header which contains segment metadata, e.g., size and memory permissions, for each segment. The loader uses this metadata to load each segment of the binary into memory. Due to the structure of the binary format, adding an empty segment header in the beginning of the binary enables selfrando to append an arbitrary amount of data. When the linker is finished, we append the TRaP info and RandoLib to the end of the binary and set the values of the empty segment header accord-

ingly ❸. Finally, we change the start address of the binary—its *entry point*—to RandoLib. Hence, after the loader loads the binary into memory, it will transfer control to RandoLib, which then performs the function permutation.

4.3 Load-time Function Permutation

RandoLib performs function permutation using the embedded TRaP info, and consists of two parts: a small helper stub and the main randomization module. The purpose of the helper stub (*RL Starter* in Figure 2) is to make all selfrando data inaccessible after RandoLib finishes. The operating system loader ❹ calls this stub, invoking RandoLib as the first step of program execution.

The function permutation algorithm proceeds in several steps. First, RandoLib generates a random order for the functions using the Fisher-Yates shuffling algorithm. Second, RandoLib uses the embedded metadata to fix all references that became invalid during the randomization. Finally, after RandoLib returns, the helper stub makes selfrando's data inaccessible ❺, and jumps to the original entry point of the binary.

While this approach might seem straightforward, we faced several technical challenges. For example, we have to consider that C++ code and certain assembly instructions require a certain alignment for every function. The Itanium C++ method pointer specification assumes that all functions are at least 2-byte aligned [43]. Further, we found that some assembly instructions are sensitive to alignment, e.g., `movdqa` which is commonly used in the implementation of cryptographic functions. We account for the alignment of C++ functions by increasing the size of the code segment by one byte per function. This allows RandoLib to maintain the least significant bit alignment of functions during copying. During our evaluation, we found that this alignment increases the file size on average by 0.3%.

Our implementation does not fully support alignment-sensitive assembly instructions, as they are not used by the TB. We can currently run programs that use such instructions by preserving the four least significant bits of function addresses during randomization. Moreover, we are working on a static analysis tool that can identify functions that contain these instructions, and mark them in the TRaP info so RandoLib can take their alignment constrains into account.

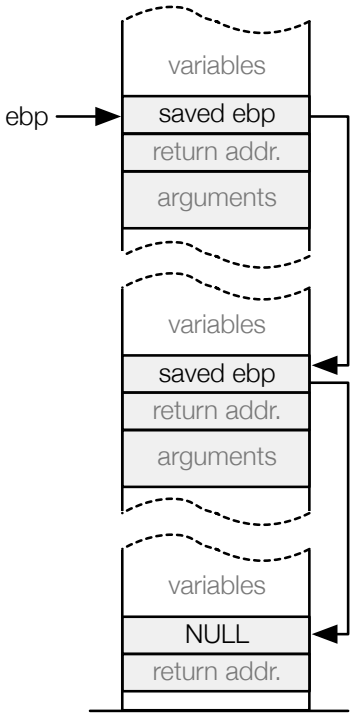

Fig. 3. Stack layout with the frame pointers.

4.4 Stack Unwinding

During program execution, the program stack is divided into *stack frames*. Each stack frame corresponds to a function call and consists of local variables, the return address, and arguments which were passed to the callee. *Stack unwinding* is the process of iterating through all active stack frames, starting from the most recent. It is mainly used for stack traces and exception handling, as both require access to previous stack frames. Exception handling uses stack unwinding to find the exception handler for a given exception after the program has thrown an instance of that exception.

Traditionally, stack unwinding is supported by chaining stack frames as a singly-linked list, where each stack frame includes a pointer to the previous stack frame. The head of the linked list is stored in a dedicated register called the *base pointer* (BP) (`ebp` on x86). When a new stack frame is added, the called function saves the BP register of the caller on the stack, then overwrites the BP register to point to the current stack frame, as shown in Figure 3.

Modern compilers omit the frame pointer for optimized code to reduce memory usage on the stack and free another register for general purpose computations. To still support stack unwinding, compilers generate additional metadata which can be used to identify individ-

ual stack frames. Function permutation invalidates function references inside the stack unwinding metadata, so RandoLib updates them.

4.5 AddressSanitizer

The TB developers use AddressSanitizer (ASan) [61] to detect memory corruption bugs in their hardened releases. To allow selfrando to be deployed on TB, selfrando needs to work correctly with ASan.

In general, selfrando does not interfere with the normal operation of ASan. When ASan detects a memory corruption, it generates a stack trace, which is supported by selfrando (cf. Section 4.4). To help troubleshoot memory corruption bugs, ASan annotates the stack trace with symbolic information. Specifically, it uses a *symbolizer* to obtain the function name and the source code location of every address in the stack trace. After selfrando randomizes the order of functions, the symbolizer can no longer correctly map the stack addresses to function names. We restore the symbolizer's ability to annotate stack traces by emitting a map file that stores the original and actual address of each randomized function. We modify the symbolizer of ASan to use the emitted mapping to map the addresses of the stack trace to the original address.

While storing the randomization map on disk is a potential security risk, exfiltrating this map would require that the attacker can read the randomization map file. The ability to read arbitrary files gives the attacker other, more significant advantages. For example, an attacker could use this advantage to disclose the full memory layout of the program by reading the special `/proc/self/mem` file.

5 Experimental Evaluation

We thoroughly evaluated selfrando from a security, performance and compatibility standpoint.

5.1 Security Analysis

We first evaluate the security of our solution and ASLR in terms of randomization entropy. This shows how well each defense resists brute force attacks. We then use a real-world exploit to compare our solution to ASLR in cases where attackers exploit information leakage which can be more effective than brute force guessing.

Randomization Entropy

For any randomization scheme the amount of entropy provided is critical, because a low randomization entropy enables an attacker to guess the randomization secret with high probability [64]. We compare selfrando to ASLR—the standard code randomization technique that is available on all modern systems.

We determined the real-world entropy of ASLR by running a simple position-independent program multiple times and analyzing the addresses, on a Debian 8.4 machine using GCC 6.1.0 and Clang 3.5.0. ASLR provides up to 9 bits of entropy on 32 bit systems and up to 29 bits of entropy on 64 bit systems. While the ASLR offset on 32 bit systems is guessable in a reasonable amount of time, such attacks become infeasible on 64 bit systems because the address space is that much larger. However, an attacker can bypass ASLR by leaking the offset that the code is loaded at in memory through a pointer into application memory. Once this offset is known the attacker can infer any address within the application, because it is used to shift the address of the whole application.

Selfrando, on the other hand, applies more fine-grained function permutation. This means the randomization entropy does not depend on the size of the address space, as it is the case for ASLR, but on the number of functions in the randomized binary. The total entropy generated by selfrando on a library containing m functions depends on the factorial of m:

$$E_t = \log_2(m!)$$

On the other hand, the attacker does not usually need to disclose the whole layout; the addresses of a few functions are enough. Assuming the attacker already bypassed ASLR, the attacker needs to disclose the least significant bits of each pointer. The entropy of a pointer to a randomized function depends on the size of the executable section s:

$$E_p = \log_2(s) - 1$$

We need to subtract 1 because the least significant bit of the addresses is preserved during the randomization. Assuming that the attacker needs gadgets in n different functions, the total number of bits the attacker needs to disclose is the minimum of E_t and n times E_p:

$$E = \min(E_t, n \times E_p)$$

In practice, E_t is much greater than E_p due to the factorial, so we can assume $E = n \times E_p$.

Technique	Entropy
ASLR (32 bit)	9 bits
ASLR (64 bit)	29 bits
Selfrando (10 KB code)	$13 \times n$ bits
Selfrando (163 KB code)	$17 \times n$ bits
Selfrando (6.5 MB code)	$22 \times n$ bits
Selfrando (92 MB code)	$26 \times n$ bits

Table 1. Randomization entropy of ASLR and selfrando for different address space sizes and function counts. For selfrando, we report the number of bits the attacker needs to guess for each function address the attacker needs.

Using TB as our model organism, we use the number of functions to calculate the minimum and maximum entropy for a binary protected by selfrando. The smallest library (`libplds4.so`) has 44 functions in 10 KB of code, while the biggest (`libxul.so`) has 242 873 functions in 92 MB. The median is 494 functions in 163 KB, while the average is 16 814 functions in 6.5 MB. Table 1 shows that for each function address, the attacker needs to guess between 13 and 26 bits. If we assume that the attacker needs the address of at least three functions, selfrando is significantly more effective than ASLR. For the smallest library, the attacker needs to guess at least 39 bits, while for the biggest, the attacker needs at least 78 bits.

Additionally, selfrando provides higher leakage resilience compared to ASLR because the attacker no longer knows the relative function layout inside each binary.

Real-world Exploits against the Tor Browser

One of our main objectives is to enhance the resilience of TB against code-reuse attacks. Previously conducted attacks, e.g., by the FBI [57], fail because these attacks do not consider selfrando (see Appendix A for an overview of the exploit the FBI used). Therefore, we analyze the attack surface of TB after selfrando was applied in a realistic attack scenario. We base our analysis on four observations we made while studying real-world exploits.

First, nearly all modern attacks exploit heap-based vulnerabilities, despite the existence of stack vulnerabilities [50]. However, whether a vulnerability can be exploited to launch a code-reuse attack depends on different factors, like how reliably the vulnerability can be triggered and the present mitigation techniques. Today, most stack-based vulnerabilities are not exploitable because they are mitigated by modern stack defenses [18, 70].

Second, information disclosure attacks are often limited to leaking heap memory because they access memory relative to the address of the vulnerable memory object. A buffer overread, for example, can be exploited to disclose consecutive memory which might contain interesting pointers, whereas a use-after-free vulnerability can be exploited to disclose interesting pointers of the freed object. In both cases the attacker is not able to (repeatedly) disclose absolute, and therefore, arbitrary, addresses. For these reasons we assume that in a practical scenario the attacker cannot leak information that is not located on the heap, e.g., stack or code pages. To overcome this limitation attackers use a technique, called *heap feng shui* [67], to place an object that contains valuable pointers near to the vulnerable object.

Third, most real-world attacks are based on ROP. While other types of code-reuse attacks exist [15, 52, 59], ROP remains the most versatile technique. To execute a ROP payload, the attacker needs to either inject his payload directly on the stack, or use a *stack-pivot* gadget to overwrite the stack pointer with an address that points to the ROP payload on the heap. As mentioned previously, the attacker usually has no access to the stack. Hence, the first gadget in the ROP chain is normally a stack-pivot.

Fourth, ROP is merely used to bypass W$\oplus$X policies and enable code injection, i.e., a small ROP payload is used to (1) mark the data memory containing the shellcode as executable and (2) branch to the shellcode. The shellcode will then perform the actual task of de-anonymizing the user or installing surveillance software. To mark a data page as executable, only a single system call is needed. Hence, the attacker requires only gadgets that load the arguments for the system call into the registers, then issue a system call and return to the shellcode.

Based on these four observations, we examined the main TB library with selfrando enabled (`libxul.so` having a size of 92MB) to find out whether an attacker is able to disclose the address of a stack-pivot and a system call gadget based on addresses that can be found on the heap. We focus on stack-pivot and system call gadgets because they are less common, and therefore, harder to disclose compared to gadgets that only load a value into a register. In total, we found ten stack-pivot and 76 system call gadgets of which only 4 and 29 respectively are available through virtual functions whose addresses are exposed on the heap through indirection tables called *virtual tables*.

We manually analyzed each function and concluded that no pointer to these functions is ever written on

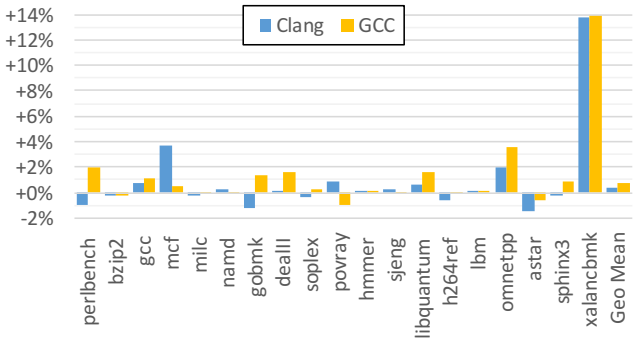

Fig. 4. Run time overhead on the benchmarks in the SPEC CPU2006 suite (full selfrando).

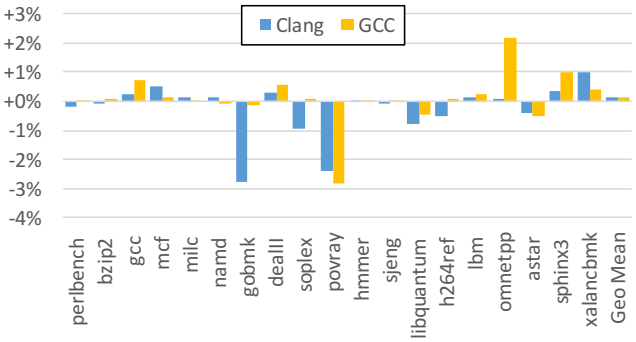

Fig. 5. Run time overhead on the benchmarks in the SPEC CPU2006 suite (identity transformation).

the heap. The reason is that these function pointers are only accessed through an indirection layer, i.e., memory objects on the heap contain a pointer to a virtual table which is located in the code or data section of the application and contains a number of pointers to virtual functions. Since the attackers can only disclose the virtual table pointer, but not the virtual table itself, as it is not on the heap, they cannot disclose gadget addresses. Note that, when only ASLR is applied, the address of the virtual table is randomized with the same offset as the ROP gadgets. Therefore, such an attack can bypass ASLR but not selfrando.

We therefore conclude that selfrando can thwart most real-world exploits. Attackers can only succeed in rare cases where they can disclose the complete heap and data section.

5.2 Performance Overhead

We performed multiple tests to measure selfrando's runtime overhead. Since selfrando works at load-time, we also measured the additional startup time.

All tests were performed on a system with an Intel Core i7-2600 CPU clocked at 3.40 GHz, with 12 GB of RAM and a 7200 RPM hard disk. We used version 5.0.3 of the Tor Browser on Ubuntu 14.04.3.

5.2.1 Load-time Overhead

We measured the load time of TB by inserting a `return` statement in the `main` function, after the dynamic libraries are loaded but before the program actually does anything. We invoked the modified program and measured the load time using the standard tool `time`. As a baseline, we used the source code of TB 5.0.3, unmodified except for the `main` function. For both versions, the reported time is the average of 10 runs. We cleaned the disk cache before each run, so the binary was loaded from the disk every time.

The average load time for the normal version was 2.046 s, while the selfrando version took 2.400 s on average. The average overhead is 354 ms. We believe this is an acceptable overhead considering the improved protection against de-anonymization attacks.

5.2.2 Run-time Overhead

To test the run-time overhead of selfrando, we ran the SPEC CPU2006 benchmark suite as well as a number of modern JavaScript benchmarks.

We executed all the C and C++ benchmarks in SPEC CPU2006 with the two standard Linux compilers (GCC and Clang) with selfrando enabled. Moreover, we ran the benchmarks with a version of selfrando that always chooses the original order for the randomization (*identity transformation*). This version runs all the load-time code but it does not actually modify the code segment. It allows us to distinguish between load-time overhead and run-time overhead. We ran each benchmark three times with the `ref` workload. The reported figures are the median values.

Figure 4 shows the performance overhead on each benchmark. The geometric mean of the positive overheads is 0.71% for GCC and 0.37% for Clang. The overhead of each benchmark except for `xalancbmk` is below 4%. We found `xalancbmk` to be an outlier, with an overhead of about 14%. We investigated this issue using the Linux performance analysis tool, `perf`, comparing the full selfrando and the identity transformation runs. We discovered a 69% increase in L1 instruction cache misses and a 521% increase in instruction

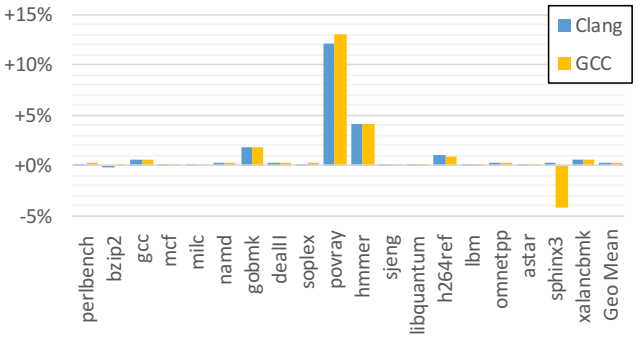

Fig. 6. Memory overhead of the benchmarks from the SPEC CPU2006 suite (full selfrando).

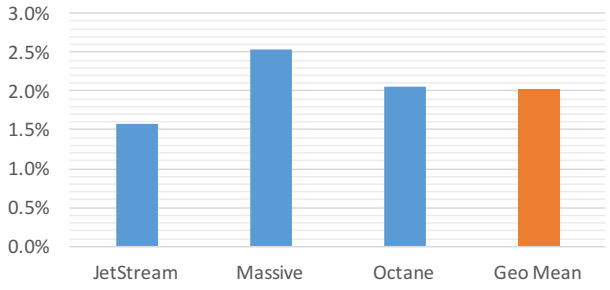

Fig. 7. JavaScript performance overhead of selfrando w.r.t. a version with all our modifications but without the actual randomization.

TLB (Translation Lookaside Buffer) misses. We believe that the `xalancbmk` benchmark is sensitive to the function layout and that some frequently executed functions must be co-located to ensure optimal performance. We didn't observe a high sensitivity to the function layout for any of the other benchmarks. A possible extension to selfrando to cope with location-sensitive programs is to automatically use performance profiling to identify groups of functions that should be moved as a single bundle similar to the work of Homescu et al. [41]. If these bundles are small enough, this extension would not significantly reduce the security of a large application (`xalancbmck` contains 13478 functions). Figure 5 shows the run time overhead with the identity transformation.

In some cases, selfrando actually improves performance. In particular, we observed that with the identity transformation the performance of `gobmk` and `povray` improves up to 2.5%. We suspect this is caused by the compiler flag that places each function in its own section, which enables further linker optimizations [31]. This flag is not enabled by default, but selfrando requires it (see Section 4.1).

Figure 6 shows the overhead on the memory usage of each benchmark. To measure the memory usage, we used the *maximum resident set size* reported by the `time` utility. The geometric mean of the positive overheads is 0.18% for GCC and 0.20% for Clang. We also measured the absolute overheads: the geometric mean of the positive values is 299 kB for GCC and 295 kB for Clang.

The memory overhead of all benchmarks except for `povray` and `hmmer` is below 2%. These benchmarks have higher relative overheads due to their small memory footprints, about 5 MB for `povray` and about 9 MB for `hmmer`. Their absolute overheads are about 600 kB and 400 kB respectively.

Finally, we evaluated selfrando with modern JavaScript benchmarks that focus on realistic web workloads: JetStream 1.1., Massive and Octane 2.0 [1–3]. As a baseline, we used a version of TB with the same modifications we need for selfrando (see Section 5.3), but without the randomization. Since selfrando does not protect JIT-compiled code, we disabled the JIT compiler by setting the *Tor Security Slider* to *Medium-High*. Figure 7 reports the results. Each benchmark produces a score (higher scores are better) and we report the relative *decrease* on the score. The geometric mean of the overheads is 2.02%, while the worst overhead is 2.5%.

Our measurements confirm that selfrando can be integrated in real-world applications with low overhead.

5.3 Compatibility

Selfrando was optimized to protect the TB which is built with GCC. However, we built several other Linux programs such as GNU Bash 4.3, GNU less 458, Nginx 1.8.0, Socat 1.7.3.0 and Thttpd 2.26. We tested each of them using application-specific workloads, such as serving files and running shell scripts, and we did not encounter any problem.

To prove compatibility to other compilers we decided to build Chromium [9]. We chose Chromium because this project has a large and complex code base, and uses Clang [47] as default compiler. Like with TB, we had to resort to the libc heap allocator, as Chromium's default heap allocator relies heavily on Thread-Local Storage (TLS) and, hence, is not fully compatible with selfrando. However, after changing the heap allocator we successfully built and run Chromium.

Both browsers implement cryptography using low-level code that embeds data in the code segment. This produces unexpected results when the data is moved

along with the functions and the alignment is not preserved. For Firefox, we disabled the low-level implementation and we used the high-level one. For Chromium, there was no easy way to disable the alignment-sensitive code and we had to preserve the four least significant bits of the addresses during the randomization (see Section 4.3).

To ensure selfrando did not break any functionality we tested both browsers with popular websites[4] and we did not encounter any problems.

5.4 Including selfrando in the Tor Browser

The Tor Project is experimenting with a number of different tools to produce *hardened builds* of TB [56]. We worked closely with their developers in order to make it easy to integrate selfrando in TB. Selfrando was added to the *nightly hardened* builds released and May 13, 2016 or later [45]. They plan to release a hardened version that includes selfrando and to evaluate the inclusion of selfrando in the normal version.

6 Discussion

Privacy Implications

Load-time code randomization effectively creates a unique code layout for each TB session. Theoretically, an adversary with the ability to read memory can exploit this to create a unique fingerprint to identify the user across different websites.

However, we argue that modern Web technologies (like JavaScript) by themselves can be exploited to leak information to identify users across different websites. Moreover, even without selfrando, an attacker that can read the memory or leak some pointers can fingerprint a browsing session in a number of different ways. ASLR creates code diversity because the binary and the libraries are loaded to different addresses. ASLR also affects the allocation of dynamic data structures such as the heap, stack and data within the heap. The allocation of these data structures is highly dependent on the usage of the browser, and hence, it is very likely that the disclosure of heap addresses is already enough to identify users. Additionally, a potential fingerprint of the randomized code is only valid for one browsing session; after a browser restart, the code layout is randomized differently. Finally, selfrando is compatible with XoM [7, 12, 20, 34] which prevents reading memory that contains code in the first place.

Hence, our randomization scheme does not increase the risk of fingerprinting.

System libraries

While software protected by selfrando works smoothly with unprotected libraries (and protected libraries work smoothly with unprotected programs), the security guarantees provided by selfrando are obviously limited to software that was re-built with selfrando. The TB includes most needed libraries, and hence, is not affected by this.

Future Work

Our current implementation focuses on applying selfrando to the TB. We are currently working on improving operating specific features, such as the support for thread-local storage (TLS). TLS is heavily used in Firefox's default heap allocator *jemalloc*, however, it is possible to build the TB using the default heap allocator provided by libc instead, which does not rely on TLS. In fact, the TB developers expressed their desire to use a different allocator as well [56].

7 Related work

Run-time defenses usually rely on either memory randomization or integrity checks to prevent vulnerability exploitation.

7.1 Randomization-based defenses

We refer to the SoK paper by Larsen et al. [46] for a thorough analysis of the proposed software diversity tools and limit our discussion to recent works which are relevant to our purposes.

XIFER by Davi et al. [22] is a load-time fine-grained randomization tool that does not require access to the source code or offline analyses. However, its processing speed (< 0.7 MB/s) makes it unsuitable for complex applications that need to be loaded quickly.

Giuffrida et al. [35] proposed a compiler-based periodic re-randomization strategy for microkernels; this

4 To get a representative set, we selected the Alexa Top 100 sites (http://www.alexa.com/topsites) of November 2015.

strategy would require end users to compile the TB locally on their system which is impractical for users with low end systems and would significantly increase the download size of the TB. Homescu et al. [42] build a compile-time randomization approach that scales to large applications such as the TB but requires that each user download a unique copy of the browser. The approaches by Giuffrida et al. and Homescu et al. both require a heavily customized compiler and do not work with the standard build tools for Linux and Windows.

Instruction Location Randomization (ILR) by Hiser et al. [40] rewrites binaries in a new randomized encoding that is interpreted by a virtual machine with a performance overhead of about 15%. Unlike our approach, ILR is incompatible with just-in-time compiled code.

Binary stirring by Wartell et al. [77] processes binaries at install time by disassembling them and adding a load-time component; it also needs a run-time component due to imperfect disassembly. It is not suitable for our purposes since it relies on a commercial disassembler that cannot be bundled with free software. Additionally, performing additional processing at installation time invalidates the code signature of a signed program.

Marlin by Gupta et al. [38] also randomizes binaries at load time. Unlike binary stirring, Marlin does not contain a runtime-component to detect and compensate for disassembly errors. While the omission of a runtime component lowers overheads in time and space, Marlin is limited to simple ELF binaries that disassemble without errors.

A recent patch submitted to OpenBSD [25] randomizes the layout of the C library during system boot. In particular, the patch permutes the linking order of each translation unit. This shuffles symbols (e.g. functions) relative to symbols defined in other files but does not change the order of symbols defined in the same translation unit. The OpenBSD approach therefore adds less entropy than selfrando which shuffles each function independently no matter what translation unit defines it. Moreover, selfrando generates a different layout for each application each time it launches, preventing the attacker from leveraging a vulnerability in one application to disclose the layout of the library in a different application on the same system.

7.2 Leakage-resilient diversity approaches

Unfortunately, security tools based solely on randomization are vulnerable to attacks aimed at disclosing the pointers to code pages. Snow et al. [66] showed that, if the attackers can read arbitrary memory pages through a vulnerability, they can recursively scan the memory, find other code pages, disassemble them and craft an ad-hoc ROP attack (JIT-ROP). Bittau et al. [8] showed that it is possible to perform a similar attack even without a complete memory read vulnerability, just by observing whether the program crashes for a particular input (this particular attack would not work if the program randomizes itself at each run).

Thus, even fine-grained randomization does not provide complete leakage resilience on its own. This has motivated numerous papers that combine memory randomization techniques with integrity checks (such as execute-only memory) to provide comprehensive protection.

Execute-only memory on x86 processors is difficult to achieve because read permissions are implicitly granted to executable pages. To do so, XnR by Backes [7] marks all pages *not present* and inspects every page reference inside the operating system page-fault handler. HideM by Gionta et al. [34] uses a particular TLB implementation available in certain processors. Readactor by Crane et al. [20] uses a lightweight hypervisor in order to enable the extended page tables feature in modern x86 processors and enforce execute-only memory in hardware. LR^2 by Braden et al. [12] uses a software-only approach based on load masking.

Many of these tools include randomization to provide comprehensive attack resilience; most implementations randomize the code at compile time. These tools could be made more practical by using selfrando to simplify distribution without sacrificing security.

7.3 Integrity-based defenses

Control-flow integrity (CFI) [4, 5] prevents control flow hijacking by only allowing jumps and calls at run-time that are present in the source. Implementing CFI with acceptable performance overhead on commodity hardware is hard; thus, many CFI implementations trade a coarse-grained CFI enforcement for better performance.

Most CFI implementations do not rely on randomization, so an attacker can exploit a coarse-grained CFI policy by carefully constructing a malicious payload offline and then using it [13, 23, 36, 37]

Finally, Code-Pointer Integrity (CPI) aims to prevent pointer hijacking by storing code pointers, pointers to code pointers etc. in a safe region; all accesses to the safe region are instrumented to ensure the integrity of the pointers. Performance overhead is relatively small

because CPI only needs to instrument a subset of memory operations. The critical issue is the protection of the safe region; on 64-bit Intel processors, segmentation is not available, thus CPI is forced to use information hiding. Unfortunately, the most efficient implementations of this defense can also be bypassed [28].

8 Conclusions

The most widely used and privacy-sensitive programs have large code bases which makes it virtually impossible to ensure that they contain no vulnerabilities. Many exploit mitigations have been proposed to prevent attacks, however no existing tool has the performance and deployability properties that are needed for complex but user-friendly software such as the Tor Browser.

We have introduced selfrando, a fast and practical load-time randomization tool. It has negligible run-time overhead, a perfectly acceptable load-time overhead, and it requires no changes to protect the Tor Browser.

Moreover, selfrando can be combined with integrity techniques such as execute-only memory to further secure the Tor Browser and virtually any other C/C++ application.

Acknowledgments

This work was supported in part by the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union's Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-15-C-0085, and FA8750-10-C-0237 and by the National Science Foundation under award number IIP-1520552.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

References

[1] Jetstream 1.1. http://browserbench.org/JetStream/.

[2] Massive: the asm.js benchmark. https://kripken.github.io/Massive/.

[3] Octane 2.0. http://chromium.github.io/octane/.

[4] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, 2005.

[5] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13, 2009.

[6] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49, 2000.

[7] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[8] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy*, 2014.

[9] Black Duck Software, Inc. Chromium project on Open Hub. https://www.openhub.net/p/chrome, 2014.

[10] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[11] E. Bosman and H. Bos. Framing signals—a return to portable shellcode. In *35th IEEE Symposium on Security and Privacy*, 2014.

[12] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium*, 2016.

[13] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium*, 2014.

[14] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium*, 2015.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM SIGSAC Conference on Computer and Communications Security*, 2010.

[16] X. Chen. ASLR bypass apocalypse in recent zero-day exploits. http://www.fireeye.com/blog/technical/cyber-exploits/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html, 2013.

[17] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12, 1993.

[18] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *8th USENIX Security Symposium*, 1998.

[19] J. Cox. Confirmed: Carnegie Mellon University attacked Tor, was subpoenaed by Feds. http://motherboard.vice.com/read/carnegie-mellon-university-attacked-tor-was-subpoenaed-by-feds, 2016.

[20] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy*, 2015.

[21] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[22] L. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Symposium on Information, Computer and Communications Security*, 2013.

[23] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, 2014.

[24] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (Just-In-Time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium*, 2015.

[25] T. de Raadt. openbsd-tech — Anti-ROP mechanism in libc. https://marc.info/?l=openbsd-tech&m=146159002802803&w=2, 2016.

[26] R. Dingledine. Tor security advisory: "relay early" traffic confirmation attack. https://blog.torproject.org/blog/tor-security-advisory-relay-early-traffic-confirmation-attack/.

[27] R. Dingledine. Tor security advisory: Old tor browser bundles vulnerable. https://lists.torproject.org/pipermail/tor-announce/2013-August/000089.html, 2013.

[28] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy*, 2015.

[29] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[30] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems*, 1997.

[31] F. S. Foundation. Gcc manual — § 3.10, options that control optimization. https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gcc/Optimize-Options.html#index-ffunction-sections-1103, 2015.

[32] M. Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, 2010.

[33] G. Fresi Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *25th Annual Computer Security Applications Conference*, 2009.

[34] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy*, 2015.

[35] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium*, 2012.

[36] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *35th IEEE Symposium on Security and Privacy*, 2014.

[37] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium*, 2014.

[38] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino. Marlin: A fine grained randomization approach to defend against ROP attacks. In *Network and System Security*. 2013.

[39] D. Herrmann, R. Wendolsky, and H. Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *ACM Workshop on Cloud Computing Security*, 2009.

[40] J. Hiser, A. Nguyen, M. Co, M. Hall, and J. Davidson. ILR: Where'd my gadgets go. In *33rd IEEE Symposium on Security and Privacy*, 2012.

[41] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automatic software diversity. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2013.

[42] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz. Large-scale automated software diversity—program evolution redux. *Dependable and Secure Computing, IEEE Transactions on*, 2015.

[43] Itanium informal industry coalition. Itanium C++ ABI: Member pointers. https://mentorembedded.github.io/cxx-abi/abi.html#member-pointers, 1999-2015.

[44] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In *22nd Annual Computer Security Applications Conference*, 2006.

[45] G. Koppen. Include selfrando patches into our hardened builds. https://trac.torproject.org/projects/tor/ticket/17406, 2015.

[46] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, 2014.

[47] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2004.

[48] C. Liebchen, M. Negro, P. Larsen, L. Davi, A.-R. Sadeghi, S. Crane, M. Qunaibit, M. Franz, and M. Conti. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[49] Microsoft. Data execution prevention (DEP). http://support.microsoft.com/kb/875352/EN-US/, 2006.

[50] Microsoft. Exploitation Trends. *Microsoft Security Intelligence Report*, 16, 2013.

[51] S. Nagy. Address sanitizer local root. http://seclists.org/oss-sec/2016/q1/363, 2016.

[52] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11, 2001.

[53] G. Owenson. Analysis of the FBI Tor malware. http://blog.owenson.me/analysis-of-the-fbi-tor-malware/, 2013.

[54] PaX Team. *Homepage of The PaX Team*, 2001. http://pax.grsecurity.net.

[55] M. Perry. Deterministic builds part one: Cyberwar and global compromise. https://blog.torproject.org/blog/deterministic-builds-part-one-cyberwar-and-global-compromise, 2013.

[56] M. Perry. iSEC partners conducts Tor Browser hardening study. https://blog.torproject.org/blog/isec-partners-conducts-tor-browser-hardening-study, 2014.

[57] K. Poulsen. FBI admits it controlled Tor servers behind mass malware attack. https://www.wired.com/2013/09/freedom-hosting-fbi/, 2013.

[58] T. Ritter and A. Grant. iSEC Partners Final Report — Tor Project Tor Browser Bundle. https://github.com/iSECPartners/publications/tree/master/reports/Tor%20Browser%20Bundle, 2014.

[59] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy*, 2015.

[60] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[61] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[62] F. J. Serna. The info leak era on software exploitation. In *Blackhat USA*, 2012.

[63] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security*, 2007.

[64] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM SIGSAC Conference on Computer and Communications Security*, 2004.

[65] sinn3r. Here's that FBI Firefox exploit for you (cve-2013-1690). https://community.rapid7.com/community/metasploit/blog/2013/08/07/heres-that-fbi-firefox-exploit-for-you-cve-2013-1690, 2013.

[66] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, 2013.

[67] A. Sotirov. Heap Feng Shui in JavaScript. In *Blackhat Europe*, 2007.

[68] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security*, 2009.

[69] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *34th IEEE Symposium on Security and Privacy*, 2013.

[70] The Clang Team. Clang 3.8 documentation SafeStack. http://clang.llvm.org/docs/SafeStack.html, 2015.

[71] The Firefox Developers. Mozilla foundation security advisory 2013-53: Execution of unmapped memory through onreadystatechange event. https://www.mozilla.org/en-US/security/advisories/mfsa2013-53/, 2013.

[72] The Gitian developers. Gitian: a secure software distribution method. https://gitian.org/.

[73] The Tor Project. The tor browser. https://www.torproject.org/projects/torbrowser.html.

[74] The Washington Post. Meet the woman in charge of the FBI's most controversial high-tech tools. http://wapo.st/1m7UMBQ, 2015.

[75] C. Tice. Improving function pointer security for virtual method dispathes. https://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=cmtice.pdf, 2012.

[76] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. W. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *14th International Symposium on Research in Attacks, Intrusions and Defenses*, 2011.

[77] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *ACM SIGSAC Conference on Computer and Communications Security*, 2012.

[78] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security Privacy*, 2009.

[79] Zerodium. Our exploit acquisition platform. https://www.zerodium.com/program.html, 2015.

A Overview of the exploit used by the FBI in 2013

In 2013, the FBI compromised a number of servers used by Tor hidden services and used them to serve an exploit to de-anonymize users of the Tor network [57]. When the user visited one of the booby-trapped pages in Tor Browser, the exploit abused an use-after-free vulnerability of Firefox in order to enable arbitrary code execution [65]. The main payload of the exploit collected the MAC address and the host name from the victim machine and sent the data to an attacker-controlled web server, bypassing Tor [53]. That message also included a unique ID provided by the booby-trapped page in order to correlate a specific user to a specific visit. The attacker then knew the public IP address, MAC address and host name of every user that visited the booby-trapped page.

JITGuard: Hardening Just-in-time Compilers with SGX

Tommaso Frassetto
CYSEC/Technische Universität Darmstadt
tommaso.frassetto@trust.tu-darmstadt.de

David Gens
CYSEC/Technische Universität Darmstadt
david.gens@trust.tu-darmstadt.de

Christopher Liebchen
CYSEC/Technische Universität Darmstadt
christopher.liebchen@trust.tu-darmstadt.de

Ahmad-Reza Sadeghi
CYSEC/Technische Universität Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Memory-corruption vulnerabilities pose a serious threat to modern computer security. Attackers exploit these vulnerabilities to manipulate code and data of vulnerable applications to generate malicious behavior by means of code-injection and code-reuse attacks. Researchers already demonstrated the power of data-only attacks by disclosing secret data such as cryptographic keys in the past. A large body of literature has investigated defenses against code-injection, code-reuse, and data-only attacks. Unfortunately, most of these defenses are tailored towards statically generated code and their adaption to dynamic code comes with the price of security or performance penalties. However, many common applications, like browsers and document viewers, embed just-in-time compilers to generate dynamic code.

The contribution of this paper is twofold: first, we propose a generic data-only attack against JIT compilers, dubbed DOJITA. In contrast to previous data-only attacks that aimed at disclosing secret data, DOJITA enables arbitrary code-execution. Second, we propose JITGuard, a novel defense to mitigate code-injection, code-reuse, and data-only attacks against just-in-time compilers (including DOJITA). JITGuard utilizes Intel's Software Guard Extensions (SGX) to provide a secure environment for emitting the dynamic code to a secret region, which is only known to the JIT compiler, and hence, inaccessible to the attacker. Our proposal is the first solution leveraging SGX to protect the security critical JIT compiler operations, and tackles a number of difficult challenges. As proof of concept we implemented JITGuard for Firefox's JIT compiler *SpiderMonkey*. Our evaluation shows reasonable overhead of 9.8% for common benchmarks.

1 INTRODUCTION

Dynamic programming languages, like JavaScript, are increasingly popular since they provide a rich set of features and are easy to use. They are often embedded into other applications to provide an interactive interface. Web browsers are the most prevalent applications embedding JavaScript run-time environments to enable website creators to dynamically change the content of the current web page without requesting a new website from the web server. For efficient execution modern run-time environments include just-in-time (JIT) compilers to compile JavaScript programs into native code.

Code-injection/reuse. Unfortunately, the run-time environment and the application that embeds dynamic languages often suffer from memory-corruption vulnerabilities due to massive usage of unsafe languages such as C and C++ that are still popular for compatibility and performance reasons. Attackers exploit memory-corruption vulnerabilities to access memory (unintended by the programmer), corrupt code and data structures, and take control over the targeted software to perform arbitrary malicious actions. Typically, attackers corrupt code pointers to hijack the control flow of the code, and to conduct *code-injection* [2] or *code-reuse* [45] attacks.

While code injection attacks have become less appealing, mainly due to the introduction of Data Execution Prevention (DEP) or writable xor executable memory (W⊕X), state-of-the-art attacks deploy increasingly sophisticated code-reuse exploitation techniques to inject malicious code-pointers (instead of malicious code), and chain together existing instruction sequences (*gadgets*) to build the attack payload [51].

Code-reuse attacks are challenging to mitigate in general because it is hard to distinguish whether the execution of existing code is benign or controlled by the attacker. Consequently, there exists a large body of literature proposing various defenses against code-reuse attacks. Prominent approaches in this context are code randomization and control-flow integrity (CFI). The goal of code randomization [34] schemes is to prevent the attacker from learning addresses of any gadgets. However, randomization techniques require extensions [5, 7, 16, 17, 24] to prevent information-disclosure attacks [18, 50, 52]. Control-flow integrity (CFI) [1] approaches verify whether destination addresses of indirect branches comply to a pre-defined security policy at run time. Previous work demonstrated that imprecise CFI policies in fact leave the system vulnerable to code-reuse attacks [8, 9, 14, 19, 25, 26, 49]. Further, defining a sufficiently accurate policy for CFI was shown to be challenging [21].

Data-only attacks. In addition to the aforementioned attack classes, *data-only* attacks [13] have been recently shown to pose a serious threat to modern software security [30]. Protecting against data-only attacks in general is even harder because any defense mechanism requires the exact knowledge of the input data and the intended data flow. As such, solutions that provide memory safety [43, 44] or data-flow integrity [10] generate impractical performance overhead of more than 100%.

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134037

JIT attacks. Existing defenses against the attack techniques mentioned above are mainly tailored towards static code making their adoption for dynamic languages difficult. For example, the JIT-compiler regularly modifies the generated native code at run time for optimization purposes. On the one hand, this requires the code to be writable, and hence, enables code-injection attacks. On the other hand, it makes state-of-the-art defenses challenging to adopt, either due to the increased performance overhead in the case of CFI [47] (+9.6%; in total 14.6%)[1], or due to unclear practicality of code-pointer hiding [16]. In particular, the authors point out that the overhead for the JIT version is much higher and not every defense deployed for static code was applied to the JIT code [16]. Further, the attacker controls the input of the JIT compiler, and can input a program that is compiled to native code containing all required gadgets. Finally, the attacker can tamper with the input of the JIT compiler to generate malicious code, as we show in Section 3.

Goals and Contributions. In this paper we present our defense, JITGuard, that hardens JIT compilers for browsers against disclosure attacks. To motivate our defense we first propose a generic data-only attack against the JIT compiler that allows to execute arbitrary code, and can bypass all existing code-injection and code-reuse defenses. Concurrently to our work, researchers published a data-only attack that targets internal data structures of Microsoft's JIT Engine [57]. As we discuss in Section 8.3 JITGuard prevents this attack as well as our DOJITA. To protect the JIT compiler against run-time attacks without relying on additional defenses like code randomization or control-flow integrity, JITGuard utilizes Intel's Software Guard Extensions (SGX) [32] to execute the JIT-code compiler in an isolated execution environment. This enables JITGuard to hide the location of JIT-code in memory while simultaneously preventing an adversary from launching data-only attacks on the JIT-compiler. In contrast to previous work we do not require expensive analysis of the generated program to construct a CFI policy [47], or synchronization between processes [54], or repetitive system calls to change memory permission [16, 41] while providing protection against data-only attacks.

To summarize, our main contributions are:

- A generic data-only attack against JIT compilers that can bypass all existing JIT code protection techniques. In contrast to a previous data-only attack [30], which only allows to manipulate data flow (e.g., to leak cryptographic keys), our attack allows to execute arbitrary code without manipulating any code pointers.
- A novel JIT compiler protection, JITGuard, which hardens JIT compilers against code-injection, code-reuse, and data-only attacks. JITGuard utilizes SGX to isolate the JIT compiler from the surrounding application. As we elaborate in Section 5 this raises a number of challenges and is technically involved.
- A proof-of-concept implementation of JITGuard for Firefox's JavaScript JIT compiler *SpiderMonkey* and real-world SGX hardware. We explain in detail how we solve several performance-related challenges that arise when executing the JIT compiler in an enclave.

[1]Compared to MCFI [46], a CFI implementation by the same author for static code.

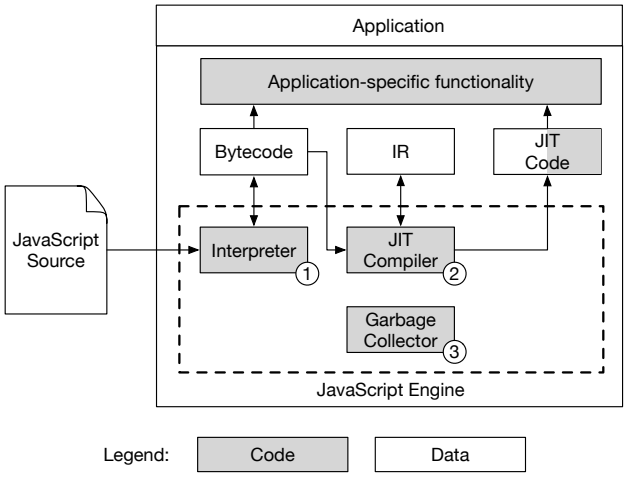

Figure 1: Main components of a JavaScript JIT engine.

- An extensive performance and security evaluation for JITGuard. We report an average overhead of 9.8% for the integrated benchmarking suites of SpiderMonkey.

2 BACKGROUND AND RELATED WORK

In this section we briefly explain the technical concepts required to understand the remainder of this paper. We start with a short introduction of Intel's Software Guard Extensions (SGX) [32] which constitutes the trusted computing base for our defense tool JITGuard. Then we explain the basic principles of just-in-time compilers for browsers, which is the main use case for our proof-of-concept implementation in this paper.

2.1 Software Guard Extensions

SGX is a hardware extension enabling isolated execution environments called *enclaves*. Enclaves are created within a user-mode process and cannot be accessed by any (higher privileged) system entity, including the creator process and the OS. This is enforced by the CPU through access control. In particular, the memory of an enclave can only be accessed by the code executed within the enclave. However, this policy can only be enforced while the enclave memory resides within the CPU-internal memory (cache). To protect enclave memory outside of the CPU, it is encrypted and integrity-protected with an enclave-specific key. The encryption prevents attackers from accessing any secrets that are stored within enclaves. Before the enclave memory is loaded into the CPU, SGX verifies its integrity to ensure that an adversary did not include any modifications.

The code executed within an enclave runs in the context of the creating process. Thus, it can access the process memory, e.g., for communicating with the host. SGX ensures that the enclave is isolated from other processes, enclaves, and the operating system.

2.2 JIT Engines

JIT engines provide a run-time environment for high-level scripting languages, allowing the script to interact with application-specific

functionality. They leverage so-called just-in-time (JIT) compilers to transform an interpreted program or script into native code at run time. Browsers in particular make heavy use of JIT compilers to increase the performance of *JavaScript* programs. JavaScript is a high-level scripting language explicitly designed for browsers to dynamically change the content of a website, e.g., in reaction to user input. In general, JIT engines consist of at least three main components, as shown in Figure 1: ① an interpreter, ② a JIT compiler and ③ a garbage collector.

① *Interpreter.* The purpose of JIT compilers is to increase the execution performance of JavaScript by compiling the script to native code. Since compilation can be costly, usually not all of the scripting code is compiled. Instead, JIT engines include an interpreter which transforms the input program into unoptimized *bytecode*, which is then executed by the interpreter. During the execution of the bytecode, the interpreter profiles the JavaScript program to identify parts (i.e., usually functions) of the code which are executed frequently (*hot code*). When the interpreter identifies a hot code path, it estimates if compilation to native code would be more efficient than continuing to interpret the bytecode. If this is the case, it passes the hot code to the JIT compiler.

② *JIT compiler.* The JIT compiler takes the bytecode as input and outputs corresponding native machine code. Similar to regular compilers, the JIT compiler first transforms the bytecode into an *intermediate representation* (IR) of the program, which is then compiled into native code, also called *JIT code*. In contrast to the bytecode, which is interpreted in a restricted environment through a virtual machine, this native code is executed directly by the processor that runs the browser application. To ensure that malicious JavaScript programs cannot harm the machine of the user, the JIT compiler limits the capabilities of the emitted JIT code. In particular, the compiled program cannot access arbitrary memory, and the compiler does not emit potentially dangerous instructions, e.g., system call instructions. Further, the emitted native code is continuously optimized, and eventually, de-optimized when the JIT compiler determines that this is not needed anymore. Because the JIT compiler has to write the emitted native code to memory as part of its output, the most straightforward way of setting up *JIT code pages* is to set them as read-write-executable. Since such pages represent an easy target for attackers, browsers started mapping JIT pages as writable while the compiler emits the native code, and re-mapping the JIT pages to non-writable afterwards [41]. However, there is still a window of opportunity for an attacker while the compiler is emitting the code.

③ *Garbage Collector.* The last major component is the garbage collector. In contrast to C and C++, in JavaScript the memory is managed automatically. This means that the garbage collector tracks memory allocations and releases unused memory when it is no longer needed.

2.3 JIT-based Attacks and Defenses

Typically attacks on JIT compilers exploit the read-write-executable JIT memory in combination with the fact that attackers can influence the output of the JIT compiler by providing a specially crafted input program. In the popular pwn2own exploiting contest, Gong [28] injected a malicious payload into the JIT memory to gain arbitrary code execution in the Chrome browser without resorting to code-reuse attacks like return-oriented programming (ROP) [51]. To prevent code-injection attacks, W⊕X was adapted for JIT code [11, 12, 16, 41]. However, as discussed in the previous section, JIT code pages must be changed to writable for a short time when the JIT compiler emits new code, or optimizes the existing JIT code. Song et al. [54] demonstrated that this small time window can be exploited by an adversary to inject a malicious payload. They propose to mitigate this race condition by splitting the JIT engine into two different processes: an untrusted process which executes the JIT code, and a trusted process which emits the JIT code. Their architecture prevents the JIT memory from being writable in the untrusted process at any point in time. Since the split JIT engine now requires inter-process communication and synchronization between the two processes, the generated run-time overhead can be as high as 50% for JavaScript benchmarks. Further, this approach does not prevent code-reuse attacks.

JavaScript

```
function foo() {
    var y = 0x3C909090 ^ 0x90909090;
}
```

Native Code

```
Address     Opcodes       Disassembly
0:          B8 9090903C   mov  eax, 0x3C909090
5:          35 90909090   xor  eax, 0x90909090
```

Unaligned Native Code

```
Address     Opcodes       Disassembly
1:          90            nop
2:          90            nop
3:          90            nop
4:          3C35          cmp al, 35
6:          90            nop
7:          90            nop
8:          90            nop
9:          90            nop
```

Figure 2: During JIT spraying the attacker exploits that large constants are directly transferred into the native code. By jumping into the middle of an instruction the attacker can execute arbitrary instructions that are encoded into large constants.

Code-reuse attacks chain existing pieces of code together to execute arbitrary malicious code. JIT engines facilitate code-reuse attacks because the attacker can provide input programs to the JIT compiler, and hence, influence the generated code to a certain degree. However, as mentioned in Section 2.2, the attacker cannot force the JIT compiler to emit arbitrary instructions, e.g., system call instructions which are required for most exploits. To bypass this restriction Blazakis [6] observed that numeric constants in a JavaScript program are copied to the JIT code, as illustrated in Figure 2: an adversary can define a JavaScript program which assigns large constants to a variable, here the result of `0x3C909090` xor `0x90909090` is assigned to the variable `y`. When the compiler transforms this expression into native code, the two constants are copied into the generated instructions. This attack is known as *JIT*

spraying and enables the attacker to inject 3-4 arbitrary bytes into the JIT code. By forcing the control flow to the middle of the `mov` instruction, the CPU will treat the injected constant bytes as an instruction and execute them.

JIT spraying can be mitigated by constant blinding, i.e., masking large constant C through xor with a random value R at compile time. The JIT compiler then emits an xor instruction to unblind the masked constant before using it ($(((C \oplus R) \oplus R = C \oplus 0 = C)$). While constant blinding indeed prevents JIT spraying it decreases the performance of the JIT code. Further, Athanasakis et al. [4] demonstrated that JIT spraying can also be performed with smaller constants, and that constant blinding for smaller constants is impractical due to the imposed run-time overhead. Recently, Maisuradze et al. [36] demonstrated a JIT-spraying attack by controlling the offsets of relative branch instructions to inject arbitrary bytes into the JIT code.

Another approach to mitigate JIT-spraying is code randomization. Homescu et al. [29] adopted fine-grained randomization for JIT code. However, similar to static code, code randomization for JIT code is vulnerable to information-disclosure attacks [52]. While Crane et al. [16] argued that leakage resilience based on execute-only memory can be applied to JIT code as well, they do not implement code-pointer hiding for the JIT code which makes the performance impact hard to estimate. Tang et al. [55] and Werner et al. [59] proposed to prevent information-disclosure attacks through destructive code reads. Their approach is based on the assumption that benign code will never read from the code section. Destructive code reads intercept read operations to the code section, and overwrite every read instruction with random data. Hence, all memory leaked by the attacker is replaced by random data, rendering it unusable for code-reuse attacks. However, Snow et al. [53] demonstrated that this mitigation is ineffective in the setting of JIT code. In particular, the attacker can use the JIT compiler to generate multiple versions of the same code by providing a JavaScript program with duplicated functions. Upon reading the code section the native code of the first function will be overwritten while the other functions are intact and can be used by the attacker to conduct a code-reuse attack.

Ansel et al. [3] designed a generic sandboxing approach based on Software-based Fault Isolation (SFI), which prevents the JIT-compiled code from modifying other parts of the program. The authors do not quote a single overhead figure, however, almost all of their benchmarks have an overhead greater than 20%.

Niu et al. [47] applied CFI to JIT code and found that it generates on average 14.4% run-time overhead and does not protect against data-only attacks which do not tamper with the control flow but manipulate the data flow to induce malicious behavior.

3 OUR DATA-ONLY ATTACKS ON JIT COMPILERS

Overview. As mentioned in the previous Section, existing JIT protections only aim to prevent code-injection or code-reuse attacks. However, in our preliminary experiments we observed that arbitrary remote code execution is feasible by means of *data-only* attacks which corrupt the memory *without* requiring to corrupt any code pointers. We implemented an experimental data-only attack against JIT compilers, coined DOJITA (Data-Only JIT Attack), that manipulates the intermediate representation (IR) to trick the JIT compiler into generating arbitrary malicious payloads. Our experiments underline the significance of data-only attacks, in the presence of defenses against control-flow hijacking, and motivate the design of our defense JITGuard. Figure 3 shows the high-level idea of DOJITA:

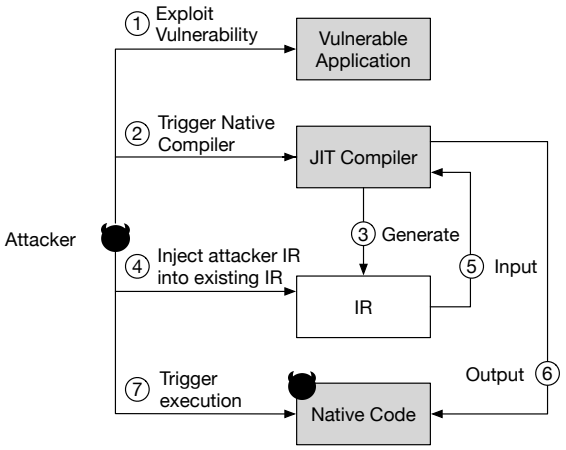

Figure 3: DOJITA enables the attacker to execute arbitrary code through a data-only attack. In particular, the attacker manipulates the IR which is then used by the JIT compiler to generate native code that includes a malicious payload.

The attacker ① exploits a memory-corruption vulnerability to read and write arbitrary data memory; ② identifies a hot function F in the input program, which will be compiled to native code; ③ during the compilation of F the JIT compiler will generate the corresponding IR; the attacker discloses the memory address of the IR in memory which is commonly composed of C++ objects; ④ injects crafted C++ objects (the malicious payload) into the existing IR. ⑤ Finally the JIT compiler uses the IR to generate the native code ⑥. Since the IR was derived from the trusted bytecode input, the JIT compiler does not check the generated code again. ⑦ Thus, the generated native code now contains a malicious payload that is executed upon subsequent invocations of the function F.

Details. For our experiments we chose the JavaScript engine of Internet Explorer, called Chakra [38]. Our goal is to achieve arbitrary code execution by exploiting a memory-corruption vulnerability without manipulating the JIT code or any code pointers. Further, we assume that the static code and the JIT code are protected against code-reuse and code-injection attacks, e.g., by either fine-grained code randomization [16], or fine-grained (possibly hardware-supported) control-flow integrity [31, 47].

For our attack against Chakra we carefully analyzed how the JIT compiler translates the JavaScript program into native code. We found that the IR of Chakra is comprised of a linked list of `IR::Instr` C++ objects where each C++ object embeds all information, required by the JIT compiler, to generate a native instruction or an instruction block. These objects contain variables like `m_opcode` to specify the operation, and variables `m_dst`, `m_src1`, and `m_src2`

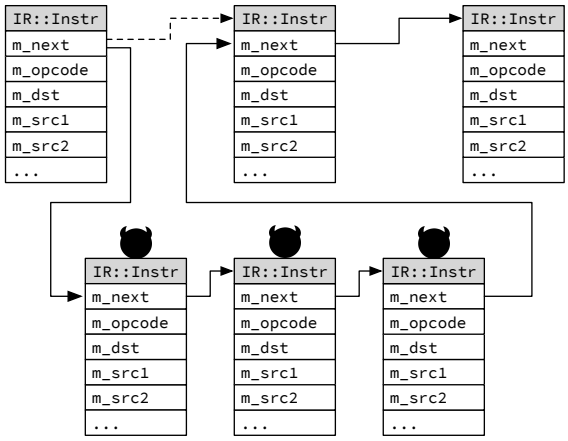

Figure 4: The IR of Chakra consists of a linked list of `IR::Instr` C++ objects. The attacker injects instructions by overwriting the `m_next` pointer of a benign object (dotted line) to point to a linked list of crafted objects.

to specify the operands for the operation. To achieve arbitrary code execution, we carefully craft our own objects, and link them together. Figure 4 shows the IR after we injected our own `IR::Instr` objects (lower part of the figure), by overwriting the `m_next` data pointer of the benign `IR::Instr` objects (upper part of the figure). When the JIT compiler uses the linked list to generate the native code it will include our malicious payload. It is noteworthy that `m_opcode` cannot specify arbitrary operations but is limited to a subset of instructions like (un-)conditional branches, memory accesses, logic, and arithmetic instructions. This allows us to generate payloads to perform arbitrary computations, and to read and write memory. However, for a meaningful attack we have to interact with the system through system calls. We inject a `call` instruction to the system call wrapper functions which are provided by system libraries. To resolve the addresses of these function, we leverage a similar approach as JIT-ROP [52]. In particular, we first disclose the address of `GetProcAddress()` which is a function that takes the name of an exported library function as an argument and returns its address. This enables our payload to resolve and call arbitrary functions, and hence, interact with the system.

Our proposed data-only attack against the JIT compiler cannot be mitigated by any state-of-the-art defenses or defenses proposed in the literature [16, 47]. The reason is that these defenses cannot distinguish the benign IR from the injected IR.

Implementation. For our proof-of-concept of DOJITA we implemented an attack framework that allows the attacker to specify an arbitrary attack payload. Our framework parses and compiles the attack payload to the ChakraCore IR, i.e., the framework automatically generates C++ memory objects that correspond to the instruction of the attack payload. Next, the framework exploits a heap overflow in `Array.map()` (CVE-2016-7190), which we reintroduced to the most recent public version of ChakraCore (version 1.4), to acquire the capability of reading and writing arbitrary memory. After disclosing the internal data-structures of the JIT compiler, we modify a number of data pointers within these structures to include our malicious IR. The JIT compiler will then iterate through the IR memory objects, and generate native code. While the injection of malicious IR into the benign IR depends on a race condition, we found that the attack framework can reliably win this race by triggering the execution of the JIT compiler repeatedly. Appendix A contains an example payload that creates a file and writes arbitrary content to it.

Our proposed data-only attack against the JIT compiler cannot be mitigated by any state-of-the-art defenses or defenses proposed in the literature [16, 47]. The reason is that these defenses cannot distinguish the benign IR from the injected IR.

In our testing, DOJITA succeeded 99% of the times.

Comparison to Related Work. Independently from our work, Theori [57] published a similar attack that also targets the internal data structures of Microsoft's JIT compiler. Their attack targets a temporary buffer which is used by the JIT compiler during compilation to emit the JIT code. This temporary buffer is marked as readable and writable. However, once the JIT compiler generated all instruction from the IR, it relocates the content of the temporary buffer into the JIT memory which is marked as readable and executable. By injecting new instructions into this temporary buffer one can inject arbitrary code into the JIT memory. Microsoft patched the JIT compiler to include a cyclic redundancy checksum of the emitted instructions during compilation. The JIT code is only executed if the checksum of the relocated buffer corresponds to the original checksum.

This defense mechanism which was recently added by Microsoft *does not* prevent our attack. While the attack by Theori [57] is similar to ours, we inject our malicious payload at an earlier stage of the compilation. As a consequence, the checksum, which is computed during compilation, will be computed over our injected IR. Since we do not perform any modifications in later stages, the checksum of the relocated buffer is still valid and the JIT compiler cannot detect our attack.

In the remainder of this paper, we present our novel defense that leverages Intel's SGX to mitigate code-injection, code-reuse, and data-only attacks against just-in-time compilers (including DOJITA).

4 THREAT MODEL AND ASSUMPTIONS

The main goal of this paper is to mitigate attacks that target JIT code generation and attacks exploiting the JIT-compiled code. Therefore, our threat model and assumptions exclude attacks on the static code. Our threat model is consistent with the related work in this area [6, 16, 36, 47, 54].

- **Static code is protected.** State-of-the-art defenses against code-injection and code-reuse attacks for static code are deployed and active. In particular, this means that code-injection is prevented by enforcing DEP [37], and code-reuse attacks are defeated by randomization-based solutions [16, 17], or (hardware-assisted) control-flow integrity [1, 31, 58]. Additionally, we assume that the static code of the application and the operating system are not malicious.
- **Data randomization.** The targeted application employs Address Space Layout Randomization (ASLR) [48]. This prevents an adversary from knowing any addresses of allocated data regions a priori and enables us to hide sensitive data from the attacker.

- **Secure initialization.** An adversary can only attack JITGuard after its initialization phase.
- **Memory-corruption vulnerability.** The target program suffers from at least one memory-corruption vulnerability. The attacker can exploit this vulnerability to disclose and manipulate data memory of *known* addresses. This is a common assumption for browser exploits [14, 49, 52].
- **Scripting Engine.** An adversary can utilize the scripting engine to perform arbitrary (sandboxed) computations at run time, e.g., adjust the malicious payload based on disclosed information.

The goal of the adversary is to gain the ability to execute arbitrary code in the browser process. The attacker can then try and further compromise the system, or leak sensitive information from the web page (e.g., launching the attack from some malicious advertisement code). The use of some defense mechanisms, like sandboxing [15, 27], can make the former attack harder. However, such defenses do not prevent the latter attack and are orthogonal to JITGuard.

We also note that any form of side-channel, e.g., cache and timing attacks to leak randomized memory addresses, or hardware attacks are beyond the scope of this paper.

5 DESIGN OF JITGUARD

Our main goal is to harden the JIT compiler against code-injection, code-reuse and data-only attacks. To achieve this we isolate all critical components of the JIT compiler from the main application, potentially containing a number of exploitable vulnerabilities. The isolation is enforced through hardware by utilizing *SGX*. Note, that intuitively one can isolate the whole JIT engine with SGX. However, the JIT code frequently interacts with static code, and since every call requires a context switch between enclave and host process, this would result in a tremendous amount of overhead. To avoid this overhead we decompose the JIT engine to execute the JIT code outside of the enclave. To prevent the attacker from exploiting the JIT code to launch code-injection or code-reuse attacks we hide the JIT code by using *randomization*. Further, we mitigate information disclosure attacks by building an indirection that transfers the control flow between the static application code and the JIT code without disclosing the address of the JIT code through *trampolines*. Figure 5 shows our design of JITGuard in more detail:

① We use SGX to isolate the JIT compiler and its data from the rest of the application. As a consequence the attacker can no longer exploit memory-corruption vulnerabilities in the host process to launch attacks against the JIT compiler, as described in Section 3. ② We randomize the JIT code and JIT stack memory addresses to protect against code-injection and code-reuse attacks and prevent the attacker from locating the JITGuard-Region. Even though our randomization does not prevent an adversary from injecting code, e.g., by compiling a specially crafted JavaScript program [6, 36], the attacker cannot disclose the address of the injected code which is required to redirect the control flow to the injected code. The same holds for code-reuse attacks where the attacker requires the addresses of the gadgets.

③ We leverage segmentation registers to build an indirection layer to prevent information-disclosure attacks that target the transition between static and JIT code. This is necessary since the attacker is able to disclose data at known addresses (see Section 4).

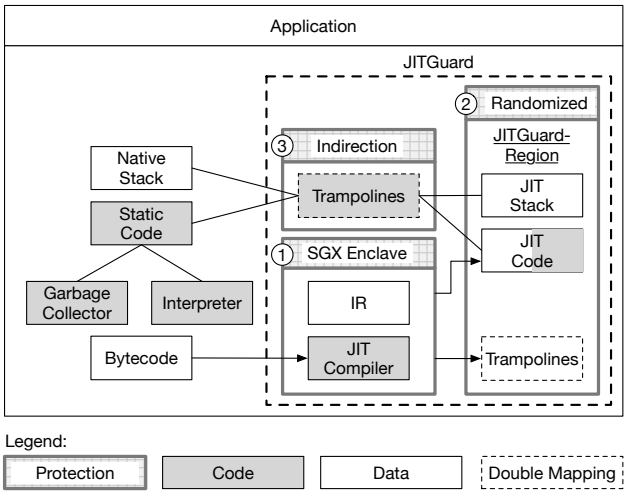

Figure 5: Design of JITGuard

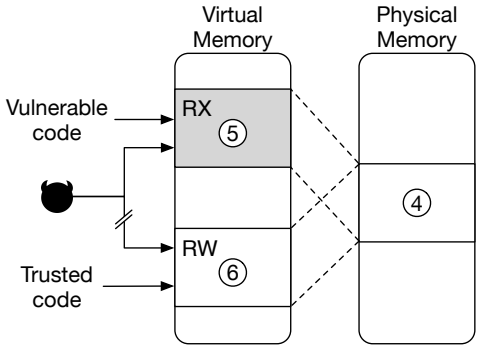

Figure 6: The same region of physical memory is mapped twice in the virtual memory with different permissions.

Thus, we utilize trampolines which contain *jump* instructions that obtain the address of the JIT code using an offset from a segmentation register. The content of the segmentation register itself is available only through a system call, hence an adversary needs to launch a successful attack against the JIT compiler to disclose it. The compiler needs to be able to efficiently update the indirection layer; however, using read-write-executable permissions would allow an attacker to simply inject new code into the trampoline mapping. To allow the former without the latter, we employ a *double mapping* of the trampolines (see Figure 6).

Using this technique, the same region in physical memory ④ is mapped twice in the virtual address space of the process. The first mapping ⑤ is executable but not writable. The second mapping ⑥ is writable but not executable, and its address is protected through randomization. The compiler uses the second mapping to update the trampolines (e.g., when a new function is compiled) and the indirection layer, while the (potentially vulnerable) static code uses the executable trampoline mapping. Although an adversary has access to the executable mapping, the address of JIT code cannot be leaked through the executable trampoline since it is protected using

the segmentation register. In the following we present a proof-of-concept implementation of JITGuard based on the JavaScript engine of Firefox called *SpiderMonkey*. We will explain in detail how we tackle the challenge of decomposing the JIT engine, adapting the JIT compiler to SGX, and preventing the JIT compiler and JIT code from leaking the location of the JITGuard-Region.

Our modifications consist of 2 673 additional lines of code, compared to 521 000 lines of C/C++ code in the SpiderMonkey source.

6 ISOLATING THE JIT COMPILER WITH SGX

The core component of JITGuard is an SGX enclave which contains the code and data of the JIT compiler and the randomization secrets. We will use *enclave* to refer to this specific enclave. While enclaves are well suited for isolating trusted code and data, the SGX threat model assumes everything outside of the enclave is untrusted. Therefore, SGX requires a context switch to execute code outside of the enclave. This is an expensive operation and makes the straightforward approach of isolating the whole JIT engine (including the generated JIT code) impractical because the JIT code frequently interacts with static application code. In particular, we measured up to 600 interactions per millisecond in our tests. However, our threat model (Section 4) is different to that of SGX: we assume that the code running outside the enclave (static code and operating system) is not malicious. This allows us to relax some of the constraints of regular enclave applications. Instead of using SGX to isolate the full JIT engine, we use it to isolate the security-critical components (JIT compiler), and to securely store the randomization secret. This approach enables us to bootstrap the JITGuard-Region, whose address is unknown to the attacker. By emitting the JIT code to the JITGuard-Region it can be executed securely outside the enclave, and we avoid disclosing the location of the JITGuard-Region by using trampolines. Thus, the JIT code can interact with the static application code without requiring SGX context switches.

In the following, we provide more details on how we initialize JITGuard and the interaction of the JIT compiler in the enclave with the rest of the JIT engine.

6.1 Initialization

JITGuard is initialized at the start of the program before the attacker can interact with the vulnerable application. Hence, we can launch the initialization phase from the static code part of the application. The initialization component of JITGuard first allocates two memory regions, the trampoline and the JITGuard-Region, and then starts the enclave.

JITGuard chooses the location of the JITGuard-Region perfectly at random and uses it to store the JIT code, the JIT stack, and the writable mapping of the trampolines. The protection of the JIT code and stack is based on the assumption that the location of the JITGuard-Region remains secret throughout the execution of the application. JITGuard achieves this by passing the randomization secret to the enclave and setting all memory that was used during the initialization phase to zero. Henceforth, all memory accesses to the JITGuard-Region are mediated through the enclave to prevent the address from being written to memory which is accessible to the attacker.

The second memory region is the executable mapping of the trampolines. This double mapping of the trampolines is necessary because JITGuard needs to modify the trampolines during run time and the attacker can infer the address of the executable trampolines based on pointers used by the static code. Without this double mapping, a less secure solution would be to switch the memory region between read-writable and read-executable. However, an adversary could still exploit the short time window while the memory is writable to inject malicious code into the trampoline region [54]. We provide more details on our trampoline mechanism in Section 7.

Finally, JITGuard sets up the JIT compiler enclave providing the address of the JITGuard-Region as a parameter. As mentioned in Section 2.2, the JIT engine consists of different components. However, we encapsulate only the JIT compiler inside an enclave. While switching between enclave and host execution has some overhead, we carefully designed JITGuard to achieve practical performance, by executing the rest of the components of the JIT engine outside the enclave. In our security analysis (Section 8) we explain how JITGuard securely interacts with the host process.

6.2 Run Time

JITGuard requires a few modifications to the JIT compiler: (1) to be compatible to SGX, (2) to prevent disclosure of the location of the JITGuard-Region, and (3) to emit the JIT code to the randomized memory region.

6.2.1 SGX Compatibility. To make the JIT compiler compatible with SGX we created a custom system call wrapper and adjusted the internal memory allocator. As mentioned in Section 2.1, the operating system is considered untrusted in the SGX design, which is why the code inside of an enclave cannot use the system call instruction. To issue a system call, the enclave code has to first switch execution to the host process, and then call a wrapper function of a system library. The SGX developer framework provides functionality to easily call outside functions from the enclave. Outside functions can then invoke any system call. However, for system calls in JITGuard we abstained from using the functions generated by the SDK for two reasons: first, the context switch function of the developer framework saves the complete state (i.e., all registers) to enclave memory and then clears the content of all registers to prevent information leakage to the host process or the operating system. This is not necessary in our case because we consider the attacker can only access application memory; second, by issuing a system call through a library function, data might be leaked outside of the enclave which then becomes accessible to the attacker. To avoid both cases, we implemented our own system call wrapper which stores the required parameters in the designated registers inside the enclave, and then exits the enclave to issue the `syscall` instruction (without storing and clearing the state or writing anything to the application memory). Further, we adjusted the internal memory allocator of the JIT compiler to use pre-allocated memory within the enclave to avoid leaking information to the application memory.

6.2.2 Leakage-resilience. Another challenge is to prevent the JIT compiler from leaking the address of the JITGuard-Region. Since the JIT compiler consists of a huge code base it is hard to verify that

no instruction leaks this address. We avoid manual inspection of the whole source code of the JIT compiler by employing a *fail-safe* technique that is based on a *fake pointer*. In particular, JITGuard converts the real pointer to the JITGuard-Region into a fake pointer by adding a random offset during the creation of the enclave. We then modify each function that requires access to the JITGuard-Region (e.g., to emit the JIT code or modify the trampoline) to first convert the fake pointer back to the original pointer. This happens as late as possible, e.g., in the very C++ statement that writes a jump target to the JIT-compiled code page. At the same time we verify that the code which uses the pointer does not leak the pointer to memory outside of the enclave. This technique is fail safe because even if a non-verified function within the enclave would leak the address, it would only leak the fake pointer. However, the fake pointer is useless to the attacker without the random offset, which is stored securely inside enclave memory.

6.2.3 JIT Code Generation. The JavaScript interpreter constantly profiles the code while it executes it. Once the profiler determines it would benefit the performance to compile the interpreted code into native code, it calls the JIT compiler. In JITGuard this requires the interpreter to issue a context switch to the enclave and to pass the interpreted code as a parameter. The advantage of this design is that we have a single point of entry for the JIT compiler. SGX allows the enclave to access the host memory, so the compiler in the enclave can directly access the data in the host memory without the need to copy the data first.

In addition to that, the JIT compiler requires a small number of functions from the host, e.g., such as timing information, for which we add dedicated enclave exit points to switch execution to the host process.

6.3 SpiderMonkey

The previously mentioned implementation details are not specific to SpiderMonkey, but are valid for most JIT compilers. In the following we discuss some SpiderMonkey-specific aspects we encountered while implementing JITGuard.

SpiderMonkey features a second JIT compiler, called IonMonkey. IonMonkey takes the native code of the regular JIT compiler, called the Baseline compiler, and speculatively optimizes it (e.g., assuming that the variables will have the same type as previous invocations). For our proof-of-concept implementation of JITGuard we disabled IonMonkey. However, from a conceptual point of view, IonMonkey can be extended in the same way as the Baseline compiler.

Further, SpiderMonkey recently adopted W⊕X for the JIT code which simplified extending SpiderMonkey with JITGuard. The reason is that JIT compilers which do not employ W⊕X expect to be able to modify the JIT code at any time, and thus modifications are spread over multiple functions. In JITGuard the native code is emitted to the JITGuard-Region, which requires us to adjust all functions that modify the JIT code. This is limited to a small number of functions in SpiderMonkey. On the other hand, JIT compilers that do not support W⊕X can be extended with JITGuard as well, although we would expect additional engineering effort because of the more widespread modifications to the JIT code.

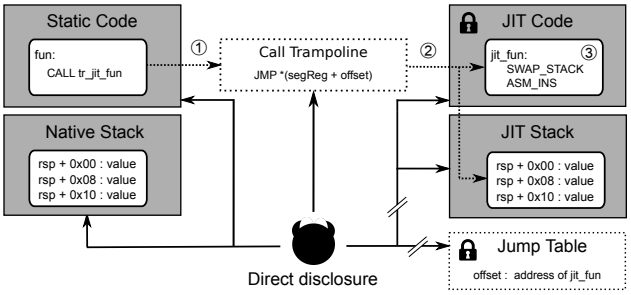

Figure 7: JITGuard mediates control-flow transfers from static code to the JIT code through call trampolines. In this way, function pointers to the randomized JIT region are hidden from an adversary.

7 TRANSFERRING CONTROL FLOW BETWEEN JIT AND STATIC CODE

JITGuard randomizes the memory location of the JIT code, JIT stack, and the writable trampoline mapping to protect them from an adversary with access to the host process memory. However, during run time the JIT code closely interacts with the static code inside the host process. Indeed, we counted the number of control-flow switches between the static code and the JIT code and measured up to 600 times per millisecond in our testing. Since the attacker has access to the host memory, we must prevent leaking any pointers from the randomized region into the non-randomized part of the host memory. This is challenging, because usually JIT and static code use the same stack during execution.

To cleanly isolate randomized JIT code from static code, we also switch to a separate stack, which is hidden inside our randomized region. In this way, the randomized stack can be used safely during JIT execution and an adversary cannot recover a return pointer to the JIT code from the native stack. In the following, we describe how JITGuard securely handles the transition from static code to JIT code execution, and JIT code to static code execution.

7.1 Static Code calls JIT Code

Static code calls JIT code functions when switching from interpreted to optimized script code. This is depicted in Figure 7.

In Step ① the static code initiates the switch to the JIT code by calling a trampoline. Each trampoline targets a single JIT code function.

If the pointers to the JIT-compiled functions were written as constants directly in the trampoline code, an adversary could easily disclose these pointers and compromise the randomized code region. To prevent this, we set up a x86 segment at initialization time[2] so that it starts at a random address. Hence, we only need to write an offset into that segment to the trampoline. In Step ② the trampoline fetches the address of the function inside the randomized area from a jump table in the randomized segment. Each trampoline consists of a single *jump* instruction that retrieves the address using a constant offset in the segment, e.g., `jmp *%gs:(0x2a00)`. The

[2]While memory segmentation is not enforced in the 64 bit modes of the x86 processor, segment registers can still be used to hold such base addresses. This is used on some operating systems, e.g., to implement fast access to per-cpu data [35]. We leverage the segmentation register `gs`, which is not used otherwise.

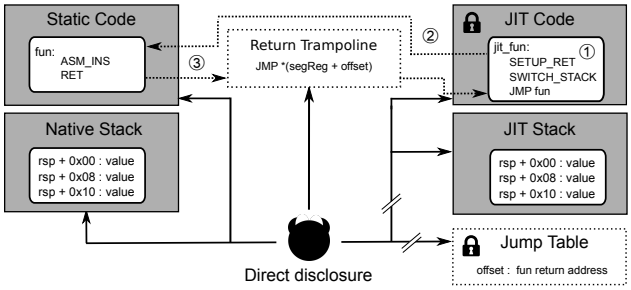

Figure 8: JITGuard mediates control-flow transfers from JIT code to static code through return trampolines. These are set up by the JIT code before jumping to the static function. This hides the return address to the JIT code from the static code.

start address of the segment cannot be disclosed by the attacker.[3] The jump table is protected from the attacker because it is located inside a randomized region.[4]

In Step ③ the JIT code switches from the native stack to the randomized stack, and subsequently starts executing its code. In particular, the randomization code updates `rsp` and `rbp` to their new location inside the randomized area and saves their previous values in the JIT stack. The JIT code expects a particular alignment of the stack, so the randomization code needs to adjust the stack to that alignment. When the JIT-compiled function returns, the randomization code restores the old values for the registers so they point to the normal stack again and returns execution to the static code.

The compiler needs a way to prepare those trampolines. If the trampolines were writable by the host code, the attacker could write malicious code to the trampoline and execute it. Thus, JITGuard leverages a double mapping of the trampolines (see also Figure 6), and keeps the address of the writable mapping hidden inside its SGX enclave, so the host code cannot read it.

7.2 JIT Code calls Static Code

During JIT code execution, it is possible to call functions inside the static code. For instance, JIT code may call a library function that is implemented in static code.

Usually, the return address of a function is stored on the stack. If the JIT code calls the native code without taking special measures, the native code can easily retrieve the return pointer from the stack and disclose the location of the JITGuard-Region. To prevent this attack, the native code uses *return trampolines* to return securely to the JIT code. Using this scheme, the return address on the native code stack actually represents the address of the return trampoline, which then retrieves the original return address using the randomized segment (see Section 7.1).

Hence, the JIT code has to prepare the return trampoline prior to calling the static code function in Step ① of Figure 8. In particular, it will store the return address to the JIT code in a jump table, that is protected because it is located inside the randomized segment. Furthermore, it will switch the stack pointer to the native stack, save the offset between the two stacks in the randomized segment, and set the return address on the native stack to point to the return trampoline.

In Step ②, the JIT code then issues the static code function call. The static code then executes normally[5] until it returns. The return trampoline in Step ③ then retrieves the original return address using the segment register and an offset into the jump table. Finally, it returns to the JIT code, which will restore the JIT stack using the saved offset and continue execution at the instruction immediately after the call to the static code.

8 SECURITY ANALYSIS

The goal of JITGuard is to mitigate code-injection, code-reuse, and data-only attacks against the JIT code. As written in our threat model (Section 4), protecting the static code, i.e., the browser and the static part of the JIT compiler, is beyond the scope of this paper and can be achieved leveraging existing defenses [1, 16, 33].

8.1 Code-injection/reuse Attacks

Both code injection and reuse techniques are used by the attacker to execute arbitrary code *after* the control flow has been hijacked. In particular, the attacker overwrites a code pointer with a malicious pointer to injected code or the first gadget of a ROP payload. However, this requires that the attacker knows the exact address of the injected code or the gadget.

JITGuard does not prevent the attacker from injecting code using techniques like JIT spraying [6, 36]. However, we prevent the attacker from disclosing the JITGuard-Region which contains the JIT code and data. As a consequence, the attacker cannot hijack any code pointers used by the JIT code, and cannot exploit the generated JIT code for code-injection or code-reuse attacks.

Next, we analyze the resilience of JITGuard against information-disclosure attacks.

8.2 Information-disclosure Attacks

The security of JITGuard is built on the assumption that the attacker cannot leak the address of the JITGuard-Region. Therefore, we carefully analyzed every component that communicates with the JITGuard-Region and analyzed them. In particular, there are seven components that interact with the randomized region, and hence, could potentially leak the randomization secret: (1) the initialization code, (2) the JIT compiler in the enclave, (3) the JIT code, (4) the trampolines, (5) the transitions between JIT and static code, (6) the garbage collector, or (7) system components. In the following we explain how JITGuard prevents information-disclosure attacks for each of these components.

[3] The base address of the segment can only be disclosed using a system call, `arch_prctl`, or using a special instruction, `rdgsbase`. Our threat model prevents the adversary from invoking that system call, since it is only used in the initialization code. The instruction `rdgsbase` has to be explicitly activated by the operating system, which is currently not even supported on Linux (and it is not used by Firefox).

[4] Theoretically, the native code could read the pointers in the randomized segment using an instruction like `mov *%gs:(0x2a00), %rax`, but the `gs` segment register is not used anywhere in the code of Firefox.

[5] Some native functions require access to the most recent stack frames on the JIT stack. We support this through copying the most important information of a small number of recent stack frames from the JIT stack to the corresponding location on the native code stack. The fields we copy do not contain pointers to the stack and we replace the address return pointers with the corresponding trampolines. We do not copy these frames back to the JIT stack, so the native code has no way to influence the JIT stack (except legitimately returning a value to the caller).

(1) Initialization code. During the initialization the JITGuard-Region is allocated through the `mmap` system call which returns the memory address. Next, the address of the JITGuard-Region is passed to the enclave, and we set all registers, local variables, and the stack memory that is used for temporarily spilling register to zero. This ensures that the address of the JITGuard-Region is not stored in memory outside of the enclave.

(2) Enclave. The first action the initialization function of the enclave takes is to obfuscate the address of the JITGuard-Region by adding a random value. Henceforth, the JIT compiler will work on the fake pointers. Note that those fake pointers are useless to an attacker without the random offset, which is stored securely inside the enclave. We identified 11 functions that require the actual address of the JITGuard-Region, e.g., to allocate memory for the JIT code stack, or to write the generated JIT code. We patch all of these functions to convert the fake pointer back to the original address as late as possible, e.g., in the very C++ statement that writes a jump target to the JIT-compiled code page. Further, we ensure that the original address is then not propagated in the data structures of the JIT compiler. Since we add this translation to the code ourselves, and it happens at the very last moment, we can verify that the address to the JITGuard-Region is never leaked by those 11 functions. Due to the large code base of the JIT compiler we cannot exclude the possibility that other functions leak the address of the JITGuard-Region to memory outside of the enclave. However, in this case these functions would only leak the fake pointer which cannot be de-obfuscated without possessing the randomization secret which is stored securely within the enclave.

(3) JIT code. The JIT code does not leak any pointers to the JITGuard-Region to attacker-accessible memory. To do this, it would need to leak either the program counter or the stack pointer to the heap. We carefully analyzed the JIT compiler and found no support for such behavior.

Another way the attacker could force the JIT code to indirectly leak an address that points into the JITGuard-Region is to generate an exception while the JIT code is executing. This would cause the operating system to store the current execution context (including instruction and stack pointers, which would both point into the JITGuard-Region) in a memory region readable by the attacker. There are two main strategies the attacker could use to trigger an interrupt: cause the JIT code to access invalid memory to trigger an exception, or use a timer to trigger a delayed interrupt. However, both strategies are infeasible. First, JavaScript is a memory-safe language, and the JIT-compiled code cannot access invalid memory. Second, the execution of JavaScript is single-threaded, and timer events are delivered synchronously, which means that the JIT code first safely exists, before a timer event, e.g., triggered by `setTimeout()`, is handled.

(4) Trampolines. Throughout the run time, the execution switches between the native code and the JIT code. As explained in the previous paragraph the JIT code cannot leak any addresses of the JITGuard-Region. We use trampolines as an indirection to prevent that any pointers to the JITGuard-Region are leaked to memory that can be disclosed by the static code. The trampolines adjust the stack pointer to point to the native or JIT stack, and change the control flow. The trampolines use a segment register as an indirection to access the JITGuard-Region to avoid leaking any addresses during this transition. Specifically, the CPU resolves the indirection using the segment register as a base address. The segment base address is set in the kernel. This translation is transparent to user mode, thus, the attacker cannot disclose the location of the JITGuard-Region through the trampolines.

(5) JIT/static code transitions. To ensure the JIT code does not leak any information when it calls a static function, we check any arguments and the CPU registers to make sure they do not represent or contain pointers to the JITGuard-Region. We use similar checks to verify the return value of JIT-compiled functions to static functions.

(6) Garbage collector. Dynamic languages employ a garbage collector for automatic memory management. This requires the garbage collector to be aware of all memory that is used throughout the execution. On the other hand, the garbage collector code outside the enclave cannot handle addresses in the JITGuard-Region. We moved the code responsible for the garbage collection of sensitive memory areas (JIT-compiled code, JIT stack) to the enclave, where the actual addresses are available. As a consequence, the attacker cannot leak addresses to the JITGuard-Region by disclosing memory used by the garbage collector.

(7) System components. Linux's proc filesystem [22] provides a special file for each process that contains information about its complete memory layout. If the attacker gains access to this file, the attacker can disclose the address of randomized memory sections, including the JITGuard-Region. However, this file is mainly used for debugging purposes and on recent versions, access requires higher privileges by default. Additionally, sandboxes, which are used as an orthogonal defense mechanism to isolate JIT engines from the rest of the system (see Section 4), prevent any access to this file.

8.3 Data-only Attacks

During a data-only attack the attacker manipulates the data on which the existing code operates. As we have shown in Section 3, attacks like DOJITA are as powerful as code-injection attacks. JITGuard mitigates data-only attacks like DOJITA by isolating the code and data of the JIT compiler in an enclave, and isolating it from the untrusted host process. Hence, the attacker can no longer manipulate the intermediate representation of the JIT compiler to launch DOJITA-like attacks. This also prevents attacks [57] that target the temporary output buffer of the JIT compiler because this buffer is within the enclave.

For this reason, the only remaining data-only attack vector on the JIT compiler is its direct input, i.e., the unoptimized JavaScript bytecode which should be compiled. However, this bytecode representation is already used by the JIT engine during interpreter execution. In Section 2.2 we explained that the interpreter limits the capabilities of the interpreted bytecode for security reasons. This is why the bytecode representation is designed in such a way, that potentially harmful instructions cannot be encoded. For instance, it does not support system call instructions, absolute addressing, unaligned jumps, or direct stack manipulation. As a consequence,

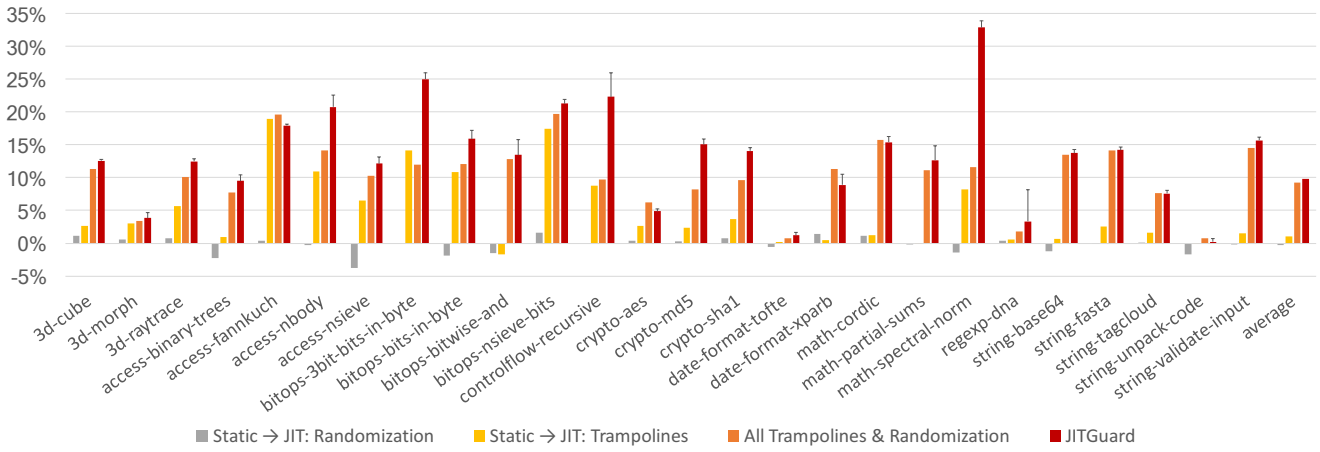

Figure 9: JavaScript performance overhead for Sunspider 1.0.2 with the various components of JITGuard enabled.

an adversary cannot utilize the bytecode to force the JIT compiler to create malicious native code, but has to resort to manipulating the IR of the JIT compiler (which is mitigated by JITGuard).

The bytecode uses integer IDs to resolve call targets, which cannot be exploited by themselves. The IDs are then resolved using tables, which an adversary could theoretically compromise using a data-only attack. However, this attack would also work in the absence of any JIT compiler, and hence, it is not directly related to JITGuard.

9 PERFORMANCE EVALUATION

We rigorously evaluated the performance impact of JITGuard on SpiderMonkey using the JavaScript benchmark Sunspider 1.0.2 [56].

Sunspider is a well-known benchmark suite that focuses on the core of the JavaScript language and is suggested by Mozilla to measure the performance of SpiderMonkey [42]. The benchmark includes multiple real-world tasks that are used in modern JavaScript apps, like dealing with JSON, code decompression, and 3D raytracing. We chose this benchmark since it only uses the core functionality of JavaScript, but it does not depend on other parts of the browser, like the DOM. Our implementation of JITGuard only includes the core JavaScript engine. The tests from the Sunspider suite are also widely used in recent browser benchmarks: as an example, the JetStream suite incorporates eleven tests from Sunspider.

Sunspider strives to be statistically sound. The total score of Sunspider is the total time needed to perform each of the benchmarks. We ran each benchmark ten times, and report the relative overhead on the weighted average of the run times, which equals the relative overhead on the total time.

We performed all evaluations on a computer with Ubuntu 14.04.4 LTS with the Linux kernel version 3.19.0.25. The machine has an Intel Core i7-6700 processor clocked at 3.40 GHz and 32 GB of RAM. We applied our modifications to SpiderMonkey version 47. To ensure the reliability of the results, we disabled the dynamic frequency scaling of the processor.

To fully understand the impact of each component of our design, we measured the overhead of each of them independently, as well as the overall impact of JITGuard. We summarize our results in Figure 9.

Static Code → JIT Randomization. First, we evaluated the randomization of the stack during the transition from static code to JIT-compiled code (*Static → JIT: Randomization* in Figure 9; see Section 6.1). This component has no measurable overhead, since we only add a small constant overhead to each call to the JIT code. *bitops-nsieve-bits* has the greatest overhead, 1.6%.

Static Code → JIT Trampolines. Second, we evaluated the impact of the trampolines that are used for calls from the static code to the JIT-compiled code (*Static → JIT: Trampolines* in Figure 9; see Section 7). The average overhead of this component is around 1.0%, since we only add one jump instruction compared to the unmodified flow. Five benchmarks in groups *access*, *bitops*, and *controlflow* have the highest overheads, ranging from 10% to 19%.

Upon investigation we found that their usage of the trampolines is significantly higher than usual, up to 316 calls per microsecond compared to the average of 83 calls per microsecond for all benchmarks.

Both Trampolines and Randomization. We then measured the impact of the trampolines and stack randomization that are employed for calls from JIT-compiled code to static code, in addition to the previous components (*All Trampolines & Randomization* in Figure 9). We measured these components together as the implementation depends on the previous components for performance reasons. The average overhead in this case is 9.2%. *access-fannkuch* and *bitops-nsieve-bits* have the highest overhead, exceeding 19%, due to their high overheads in the previous test (18%). *bitops-bitwise-and* and *math-cordic* have the highest additional overhead w.r.t. the previous tests, moving from below 2% to 12.9% and 15.7% respectively. This additional overhead is due to their high frequency of calls from the JIT code to the static code, 579 and 594 times per millisecond respectively, compared to the average of 196 times per millisecond for all benchmarks. This overhead is due to the imbalance between *call* instructions and *ret* instructions, which thrashes the processor's return stack. This is necessary to implement our security guarantees.

The additional overhead of other benchmarks is correlated with the frequency of these transitions as well.

Full JITGuard. We then measured the impact of the full JITGuard (*Full JITGuard* in Figure 9, where the error bars refer to the 95% confidence interval on the values). The average overhead for the complete scheme, including trampolines, stack randomization, and SGX compiler, is 9.8%, implying that the overhead due to SGX communication and SGX mode switches is well below 1%. This overhead specifically related to SGX is due to the low number of calls to the SGX compiler. In average, the SGX compiler is called only 6 times for each benchmark, while the maximum number of calls is 23. The maximum overhead in this benchmark is *math-spectral-norm*, which exceeds 32%. However, the overhead is still just 4.8 ms in this case; the higher relative overhead is due to the very fast run time of this benchmark, 14.6 ms compared to the average of 230 ms.

Finally, we compared our results to another run of the benchmark, with all JIT compilers disabled (interpreter only). JIT allows the benchmark to run more than 13 times faster on average and up to 260 times faster for some benchmarks. This confirms that JIT-compiled code is one order of magnitude faster than the interpreter, even including our overhead of 9.8%.

10 DISCUSSION

Portability of JITGuard. Applying JITGuard to a JIT engine requires manual effort. However, we argue this one-time effort scales due to the similarity in the high-level design of major JIT engines and their limited number. In fact, other mitigations, like CFI [31, 39, 58], require individual effort for each JIT engine as well.

Choice of different JavaScript Engines. The attentive reader may have noticed that our attack was implemented for Edge's JIT engine while our defense hardens Firefox's JIT engine. This is due to the fact that we started both projects independently from each other. However, the general idea of both the attack and the defense leverage design features which are common to all major JIT engines and are, thus, general.

Effectiveness of memory hiding. A number of recent works [20, 23] have questioned the effectiveness of memory hiding to protect sensitive memory areas that are not referenced elsewhere in memory. Gawlik et al. [23] specifically consider a web browser and introduce *crash-resistant programming*. However, one of the countermeasures they mention, *guard pages*, can be successfully applied to JITGuard since it only has one randomized region that needs to be protected. Gawlik et al. exploit signal handlers as an *oracle* in order to disclose whether a specific page is mapped. The code of those handlers can be augmented so that it calls a specific entry point on the enclave every time such an exception happens. If the address where the signal happened is close to or inside the JITGuard-Region, the enclave will then immediately terminate the program before the address can be exploited by the malicious code.

Alternative Techniques. To isolate the JIT compiler one could use randomized segments protected through segment registers, or a separate process. Using the randomized segments to hide the compiler, its stack, and its heap would be possible, but would require a considerable effort to make sure that no information leak is possible. On the other hand, SGX provides a clean separation.

Existing browsers can be retrofitted with an SGX-based design, since it preserves the synchronous call semantics of existing code. Using a separate process for the compiler, instead, requires a substantial redesign to support the asynchronous communication used in IPC.[6] Using separate processes also means the processes would have different address spaces and, thus, a higher overhead would be required due to additional communication and synchronization. Moreover, a remote procedure call from the browser to the separate compiler process would incur additional latency if that process is not already running on another core, which is unlikely, especially in case of elevated system load. On the other hand, the SGX enclave is executed on the same core, so it does not require any action from the system scheduler to run. The enclave can also leverage the data already stored in the CPU caches. In our evaluation, the overhead due to SGX is well below 1%. Finally, the remote attestation capabilities of SGX can be leveraged to prove to the server that the browser is using the JITGuard compiler and that it was not tampered with.

11 CONCLUSION

Protection of modern software against run-time attacks (code injection and code reuse) has been a subject of intense research and a number of solutions have been deployed or proposed. Moreover, recently, researchers demonstrated the threat of the so-called data-only attacks that manipulate data flows instead of the control flow of the code. These attacks seem to be very hard to prevent because any defense mechanism requires the exact knowledge of the input data and the intended data flow. However, on the one hand, most of the proposed defenses are tailored towards statically generated code and their adaption to dynamic code comes with the price of security or performance penalties. On the other hand, many widespread applications, like browsers and document viewers, embed just-in-time compilers to generate dynamic code.

We present a generic data-only attack, dubbed DOJITA, against JIT compilers that can successfully execute malicious code even in the presence of defenses against control-flow hijacking attacks such as control-flow integrity (CFI) or randomization-based defenses. We then propose JITGuard, a novel defense to mitigate code-injection, code-reuse, and data-only attacks against just-in-time compilers (including DOJITA). For this we utilize Intel's Software Guard Extensions (SGX), and explain the challenges that we needed to tackle. As proof-of-concept we implemented and evaluated JITGuard for Firefox's JIT compiler *SpiderMonkey*. The average overhead for the complete scheme, including trampolines, stack randomization, and SGX compiler, is 9.8%, where the overhead due to SGX communication and mode switches is below 1%. While we are working on further performance optimizations, our prototype already demonstrates practicality of JITGuard.

[6] Recent versions of Chakra have been redesigned [40] around an out-of-process compiler. Their defense required 27 000 additional lines of code, compared to 640 000 lines of C/C++ code in the Chakra source.

ACKNOWLEDGMENTS

This work was supported in part by the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union's Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[2] Aleph One. 2000. Smashing the Stack for Fun and Profit. *Phrack Magazine* 49 (2000).

[3] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff Biffle, and Bennet Yee. 2011. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[4] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. 2015. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*.

[5] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[6] Dion Blazakis. 2010. Interpreter exploitation: Pointer inference and JIT spraying. In *Blackhat DC (BH DC)*.

[7] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*.

[8] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium (USENIX Sec)*.

[9] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Sec)*.

[10] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[11] Ping Chen, Yi Fang, Bing Mao, and Li Xie. 2011. JITDefender: A Defense against JIT Spraying Attacks. In *26th International Information Security Conference (IFIP)*.

[12] P. Chen, R. Wu, and B. Mao. 2013. JITSafe: a framework against Just-in-time spraying attacks. *IET Information Security* 7, 4 (2013).

[13] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats.. In *14th USENIX Security Symposium (USENIX Sec)*.

[14] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Marco Negro, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[15] Jonathan Corbet. 2012. Yet another new approach to seccomp. https://lwn.net/Articles/475043/. (2012).

[16] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *36th IEEE Symposium on Security and Privacy (S&P)*.

[17] Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[18] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*.

[19] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Sec)*.

[20] Isaac Evans, Samuel Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *36th IEEE Symposium on Security and Privacy (S&P)*.

[21] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howeard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[22] Roger Faulkner and Ron Gomes. 1991. The Process File System and Process Model in UNIX System V.. In *USENIX Technical Conference (ATC)*.

[23] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*.

[24] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy (CODASPY)*.

[25] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *35th IEEE Symposium on Security and Privacy (S&P)*.

[26] Enes Göktas, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *23rd USENIX Security Symposium (USENIX Sec)*.

[27] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. 1996. A Secure Environment for Untrusted Helper Applications. In *6th USENIX Security Symposium (USENIX Sec)*.

[28] Guang Gong. 2016. Pwn a Nexus Device With a Single Vulnerability. https://cansecwest.com/slides/2016/CSW2016_Gong_Pwn_a_Nexus_device_with_a_single_vulnerability.pdf. (2016).

[29] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2013. Librando: transparent code randomization for just-in-time compilers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[30] Hong Hu, Shweta Shinde, Adrian Sendroiu, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *37th IEEE Symposium on Security and Privacy (S&P)*.

[31] Intel. 2016. Control-flow Enforcement Technology Preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. (2016).

[32] Intel. 2016. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx. (2016).

[33] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[34] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *35th IEEE Symposium on Security and Privacy (S&P)*.

[35] Linux Foundation. 2014. This-CPU Operations. http://lxr.free-electrons.com/source/Documentation/this_cpu_ops.txt. (2014).

[36] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2016. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *25th USENIX Security Symposium (USENIX Sec)*.

[37] Microsoft. 2006. Data Execution Prevention (DEP). http://support.microsoft.com/kb/875352/EN-US/. (2006).

[38] Microsoft. 2015. ChakraCore. https://github.com/Microsoft/ChakraCore. (2015).

[39] Microsoft. 2015. Control Flow Guard. http://msdn.microsoft.com/en-us/library/Dn919635.aspx. (2015).

[40] Matt Miller. 2017. Mitigating arbitrary native code execution in Microsoft Edge. https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/. (2017).

[41] Mozilla. 2015. W xor X JIT-code enabled in Firefox. https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox. (2015).

[42] Mozilla. 2016. JavaScript:New to SpiderMonkey. https://wiki.mozilla.org/JavaScript:New_to_SpiderMonkey#Benchmark_your_changes. (2016).

[43] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[44] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management (ISMM)*.

[45] Nergal. 2001. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine* 11 (2001).

[46] Ben Niu and Gang Tan. 2014. Modular Control-flow Integrity. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[47] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[48] PaX. 2003. PaX Address Space Layout Randomization. (2003).

[49] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symposium on Security and Privacy (S&P)*.

[50] Fermin J. Serna. 2012. The Info Leak Era on Software Exploitation. In *Blackhat USA (BH US)*.
[51] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[52] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *34th IEEE Symposium on Security and Privacy (S&P)*.
[53] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. 2016. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In *37th IEEE Symposium on Security and Privacy (S&P)*.
[54] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. 2015. Exploiting and Protecting Dynamic Code Generation. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*.
[55] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[56] The WebKit team. 2013. SunSpider 1.0.2. https://www.webkit.org/perf/sunspider/sunspider.html. (2013).
[57] Theori. 2016. Chakra JIT CFG Bypass. http://theori.io/research/chakra-jit-cfg-bypass. (2016).
[58] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Sec)*.
[59] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.

A EXAMPLE PAYLOAD

The text of an example payload to the framework described in Section 3 follows. Specifically, this payload creates a file and writes arbitrary content to it. This payload is parsed by our attack framework, which then creates one or more malicious IR objects for each statement. The JIT compiler then generates native code corresponding to the payload.

```
var payload = `;
  push rbp
  mov rbp, rsp
  sub rsp, 0x500
  ;
  ; Resolve function addresses
  ;
  ; LoadLibraryEx(kernel32.dll, 0,0)
  ;
  xor r8, r8
  xor rdx, rdx
  mov rcx, #addr_buf_kernel32dll
  call #addr_LoadLibraryExA
  mov [#addr_handle_kernel32], rax
  ;
  ;
  ; GetProcAddr(hKernel, CreateFile)
  ;
  mov rcx, rax
  mov rdx, #addr_buf_CreateFileA
  call #addr_GetProcAddr
  mov [#addr_ptr_CreateFileA], rax
  mov rcx, rax
  ;
  ;
  ; GetProcAddr(hKernel, WriteFile)
  ;
  mov rcx, [#addr_handle_kernel32]
  mov rdx, #addr_buf_WriteFile
  call #addr_GetProcAddr
  mov [#addr_ptr_WriteFile], rax
  ;
  ;
  ; GetProcAddr(hKernel, GetTempPath)
  ;
  mov rcx, [#addr_handle_kernel32]
  mov rdx, #addr_buf_GetTempPath
  call #addr_GetProcAddr
  mov [#addr_ptr_GetTempPath], rax
  ;
  ;
  ; GetProcAddr(hKernel, CLoseHandle)
  ;
  mov rcx, [#addr_handle_kernel32]
  mov rdx, #addr_buf_CloseHandle
  call #addr_GetProcAddr
  mov [#addr_ptr_CloseHandle], rax
  ;
  ;
  ; GetProcAddr(hKernel, ExitThread)
  ;
  mov rcx, [#addr_handle_kernel32]
  mov rdx, #addr_buf_ExitThread
  call #addr_GetProcAddr
  mov [#addr_ptr_ExitThread], rax
  ;
  ;
  ; GetTempPath()
  ;
  mov rcx, 0x400
  mov rdx, #addr_buf_1024
  call [#addr_ptr_GetTempPath]
  ;
  ;
  ; strcat(tmppath, filename)
  ;
  mov rsi, #addr_buf_file_name
  mov rdi, #addr_buf_1024
  add rdi, rax
  xor rcx, rcx
  L_strcat:
     xor rax, rax
     mov al, [rsi]
     mov [rdi], rax
     add rcx, 0x1
     add rsi, 0x1
```

```
    add rdi, 0x1
    cmp rcx, #len_file_name
    jne L_strcat
;
;
; CreateFile()
;
mov rax, rsp
add rax, 0x20
mov [rax], 0x2
add rax, 0x8
mov [rax], 0x80
add rax, 0x8
mov [rax], 0x0
xor r9, r9
xor r8, r8
mov rdx, 0x40000000
mov rcx, #addr_buf_1024
call [#addr_ptr_CreateFileA]
mov [#addr_handle_file], rax
;
;
; WriteFile()
;
mov rax, rsp
add rax, 0x20
mov [rax], 0x0
mov r9, #addr_buf_nbw
mov r8, #len_file_content
mov rdx, #addr_buf_file_content
mov rcx, [#addr_handle_file]
call [#addr_ptr_WriteFile]
;
;
; CloseHandle()
;
mov rcx, [#addr_handle_file]
call [#addr_ptr_CloseHandle]
xor rcx, rcx
call [#addr_ptr_ExitThread]
;`;

var args = {
    "#addr_LoadLibraryExA"      :
        LoadLibraryEx.hex(),
    "#addr_GetProcAddr"         :
        GetProcAddr.hex(),
    "#addr_buf_kernel32dll"     :
        addr_buf_kernel32dll.hex(),
    "#addr_handle_kernel32"     :
        addr_handle_kernel32.hex(),
    "#addr_buf_CreateFileA"     :
        addr_buf_CreateFileA.hex(),
    "#addr_ptr_CreateFileA"     :
        addr_ptr_CreateFileA.hex(),
    "#addr_buf_WriteFile"       :
        addr_buf_WriteFile.hex(),
    "#addr_ptr_WriteFile"       :
        addr_ptr_WriteFile.hex(),
    "#addr_buf_CloseHandle"     :
        addr_buf_CloseHandle.hex(),
    "#addr_ptr_CloseHandle"     :
        addr_ptr_CloseHandle.hex(),
    "#addr_buf_GetTempPath"     :
        addr_buf_GetTempPath.hex(),
    "#addr_ptr_GetTempPath"     :
        addr_ptr_GetTempPath.hex(),
    "#addr_buf_ExitThread"      :
        addr_buf_ExitThread.hex(),
    "#addr_ptr_ExitThread"      :
        addr_ptr_ExitThread.hex(),
    "#addr_buf_1024"            :
        addr_buf_1024.hex(),
    "#addr_buf_file_name"       :
        addr_buf_file_name.hex(),
    "#len_file_name"            : u64(0,
        file_name.length + 1).hex(),
    "#addr_handle_file"         :
        addr_handle_file.hex(),
    "#addr_buf_nbw"             :
        addr_buf_nbw.hex(),
    "#len_file_content"         : u64(0,
        file_content.length).hex(),
    "#addr_buf_file_content"    :
        addr_buf_file_content.hex(),
}
```

VoiceGuard: Secure and Private Speech Processing

Ferdinand Brasser[1], *Tommaso Frassetto*[1], *Korbinian Riedhammer*[2], *Ahmad-Reza Sadeghi*[1], *Thomas Schneider*[1], *Christian Weinert*[1]

[1]Technische Universität Darmstadt, Germany
[2]University of Applied Sciences Rosenheim, Germany

{ferdinand.brasser, tommaso.frassetto, ahmad.sadeghi}@trust.tu-darmstadt.de, korbinian@ieee.org, thomas.schneider@cs.tu-darmstadt.de, christian.weinert@crisp-da.de

Abstract

With the advent of smart-home devices providing voice-based interfaces, such as Amazon Alexa or Apple Siri, voice data is constantly transferred to cloud services for automated speech recognition or speaker verification.

While this development enables intriguing new applications, it also poses significant risks: Voice data is highly sensitive since it contains biometric information of the speaker as well as the spoken words. This data may be abused if not protected properly, thus the security and privacy of billions of end-users is at stake.

We tackle this challenge by proposing an architecture, dubbed *VoiceGuard*, that efficiently protects the speech processing task inside a trusted execution environment (TEE). Our solution preserves the privacy of users while at the same time it does not require the service provider to reveal model parameters. Our architecture can be extended to enable user-specific models, such as feature transformations (including fMLLR), i-vectors, or model transformations (e.g., custom output layers). It also generalizes to secure on-premise solutions, allowing vendors to securely ship their models to customers.

We provide a proof-of-concept implementation and evaluate it on the *Resource Management* and *WSJ* speech recognition tasks isolated with Intel SGX, a widely available TEE implementation, demonstrating even real time processing capabilities.

Index Terms: speech recognition, privacy protection, cloud computing

1. Introduction

Devices providing voice-based interfaces are omnipresent in today's world. Amazon Alexa, Apple Siri, Google Assistant, or Microsoft Cortana are available to the more than two billion smartphone users in 2018. Also, there is a steadily increasing number of smart-home devices, like Amazon Echo, Apple HomePod, or Google Home, solely relying on voice-based interaction. Possible application scenarios are not restricted to the consumer market but increasingly cover professional activities, for example enterprise-ready smart assistants guiding through complicated business processes in order to increase productivity.

In any of the aforementioned cases, voice data is constantly transferred to the cloud for remote speech processing, such as automated speech recognition (ASR) or speaker verification. This poses significant security and privacy risks since voice data contains sensitive biometric information as well as the spoken words: in case unprotected voice data gets out of hand, it may be abused, e.g., for impersonation attacks, assembling fake recordings, or simply extracting intimate as well as secret and sensitive content.

A naive solution to these problems is to ship the speech processing code together with corresponding models to the users to run locally. While this might be infeasible for low-end devices anyhow, it also contradicts the business interests of vendors providing such models which represent their intellectual property.

Attempts based on purely cryptographic solutions, i.e., homomorphic encryption (HE) or secure multi-party computation (SMPC), guarantee that neither user nor vendor need to reveal their respective inputs in the clear. However, as we elaborate in our review of related work in §2, these solutions are highly impractical due to their massive overhead in computation time and communication costs. Besides, none of the existing solutions considered user-specific models, i.e., the common practice to train or adapt a separate model for each user that covers deviations from the model to incorporate specific characteristics, e.g., in dialect and pronunciation.

Goals and Contributions. To overcome these limitations, we propose *VoiceGuard* in §5, an architecture that efficiently protects speech processing tasks using a trusted execution environment (TEE). It allows the secure processing of confidential data even in a hostile environment by combining cryptographic techniques with hardware-enforced code and data isolation.

Although the concept of TEEs has been known for many years, they only recently became widely available with Intel's introduction of Software Guard Extensions (SGX). SGX is Intel's implementation of a TEE available in most of their recent CPUs. It generated large interest in both academic research and industry: Signal, for example, a popular instant messaging service similar to WhatsApp, employs Intel SGX to identify the contacts in a new user's address book that are signed up to the service while all other contacts remain private [1]. The deployment of such privacy-preserving services is also facilitated by leading cloud service providers (e.g., Microsoft Azure [2]) making this CPU feature available to customers.

VoiceGuard enables secure and private speech processing, independent of who actually controls the machine performing the computation. Thus, it could be hosted by the vendor of the speech processing software, a third party service provider, or even the user. The latter on-premise solution could be preferred if it is necessary to comply to certain legal regulations or the user wants to exclude the possibility of a malicious party performing sophisticated hardware attacks.

The architecture of VoiceGuard can easily be extended to enable user-specific models, such as feature transformations (including fMLLR), i-vectors, or model transformations (e.g., custom output layers). We present a fully functional prototype implementation of VoiceGuard for ASR based on the kaldi toolkit [3]. Moreover, we conduct an empirical performance evaluation of the *Resource Management* and *WSJ* speech recognition tasks in §6, thereby demonstrating that the overhead induced by our protection measures is low enough to enable privacy-preserving speech recognition in real time.

2. Related Work

In the following, we briefly review general approaches for privacy-preserving machine learning (grouped by the underlying technology) that could be adapted to speech processing tasks which depend on the evaluation of neural networks. Furthermore, we review specialized approaches for various privacy-preserving speech processing tasks.

2.1. Privacy-Preserving Machine Learning

Secure Multi-Party Computation (SMPC). SMPC enables two or more parties to jointly compute a publicly known function without revealing private inputs to each other by executing an interactive cryptographic protocol. Recently, SMPC protocols and frameworks have been applied to both privacy-preserving training of neural networks [4] and corresponding inference [5, 6, 7, 8, 9], mostly for image classification tasks. However, compared to unprotected data processing, SMPC-based solutions require several orders of magnitude higher computation time and communication cost. They are especially impractical for on-the-fly processing due to repeated initialization costs.

Homomorphic Encryption (HE). HE allows performing operations on encrypted data s.t. the decryption of the computation result equals the outcome when performing the same operations on plaintext data. Microsoft CryptoNets [10] was the first attempt to utilize HE for secure evaluation of neural networks, followed by an improvement named CryptoDL [11], which replaces complex activation functions with approximated low-degree polynomials. Nevertheless, the reported performance results indicate that solutions based on heavyweight HE are currently far from suitable for speech recognition in real time.

TEE. SMPC via TEEs has been proposed in [12, 13, 14]. Ohrimenko et al. [15] adapt several machine learning algorithms, including neural networks, to prevent cache-based side-channel attacks in scenarios where multiple institutions use Intel SGX to securely share their datasets for training and evaluation of joint machine learning models. In [16], the authors introduce a similar protection mechanism that is efficient enough for real-time data processing: instead of preventing memory accesses that depend on sensitive data, they add noise to memory traces by accessing dummy data. The very recent Chiron [17] system allows a user to train a model using the computing resources of a cloud service provider while the training data remains hidden and the resulting model can only be accessed as a black box. This machine learning as-a-service (MLaaS) concept differs from our scenario where we assume vendors who provide existing models which should only be evaluated obliviously.

2.2. Privacy-Preserving Speech Processing

Pathak et al. [18] explored how to use the previously mentioned SC and HE techniques for privacy-preserving versions of speech processing tasks such as speech recognition and speaker verification. However, with their prototype implementation based on the Paillier HE scheme, it takes more than 3 hours to encrypt 1 s of audio and to recognize a single word out of a 10 word vocabulary. Admitting the impracticality of this approach, the authors furthermore propose a very efficient solution based on secure string-matching. Unfortunately, this approach can only be used for certain tasks such as speaker verification.

Recently, Glackin et al. [19] proposed an architecture for finding outsourced (encrypted) speech documents that contain given keywords. The architecture works as follows: (I) the client translates audio to phonetic symbols using a CNN-based acoustic model, (II) the encrypted phones and a search index are sent to a server, and (III) the server uses a searchable encryption scheme to deliver outsourced data matching the given keywords. However, this approach requires the vendor to hand the acoustic model to the user in the clear.

3. Background

For the remainder of the paper, we assume familiarity with state-of-the-art speech processing pipelines and restrict the background to the introduction of Intel SGX.

Intel SGX. Intel Software Guard Extensions (SGX) enables processing of confidential data on untrusted systems [20, 21, 22, 23]. SGX introduces the concept of *enclaves*, which are programs executed in isolation from *all* other software on a system, including privileged software, like the operating system (OS) or a hypervisor.

Enclaves are loaded as part of a host process and are embedded in its virtual memory, like a library. The initial content of an enclave is loaded from unprotected memory, hence, it can be manipulated and is not kept confidential. Therefore, confidential data must be provisioned to an enclave over a secure channel *after* it has been created. However, to ensure that secret data is not sent to a malicious (or maliciously modified) enclave, the integrity and authenticity of an enclave needs to be verified before provisioning secret data. To enable this, SGX provides a security service called remote attestation (RA). With RA, an external party can verify whether an enclave was created correctly, i.e., a cryptographic hash of the initial memory state of an enclave is signed by the platform signing key which is built into the CPU.

Once available inside an enclave, secret data can be encrypted using an enclave-specific key and written to untrusted storage, e.g., the hard disk. This *sealing* mechanism allows an enclave to use secret data across multiple instantiations.

4. Model and Assumptions

In this paper we consider a setting where three parties collaborate to perform secure and private speech processing:

(1) The *user* provides the voice data to be processed. She is concerned about her privacy and does not want the other parties to identify her based on biometric characteristics in her input. Additionally, the user does not want to reveal the content of her input to the other parties, i.e., they should not be able to access the voice data or the processing results. Lastly, the user does not want to be traceable across multiple sessions.

(2) The *vendor* provides the software required for speech processing together with corresponding models. This data constitutes the vendor's intellectual property, hence it must be kept confidential from the other parties.

(3) The *service provider* carries out the actual computations based on the user's and the vendor's inputs. The service provider could be an independent third party, e.g., a cloud service provider. Without loss of generality, the service provider could also be under the control of the user or the vendor.

Adversary Model. The adversary's goal is to extract sensitive information, i.e., the intellectual property of the vendor, the input of the user, or data that allows the adversary to identify or track the user.

We assume that the adversary is in control of the service provider's infrastructure, in particular, all computer systems involved in performing the speech processing task. The adversary has full control over the software in the service provider's infrastructure, including privileged software like the OS or a

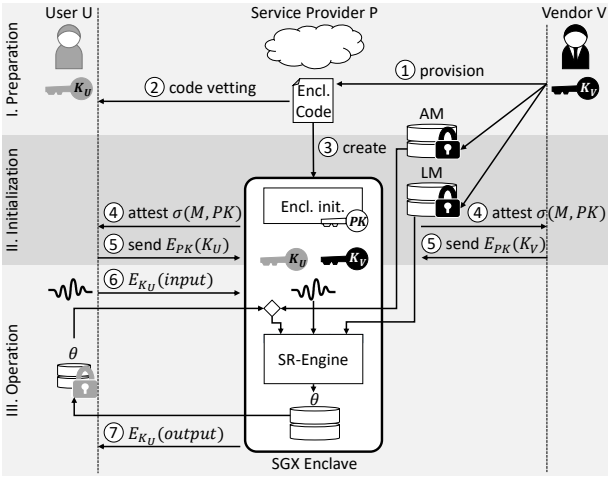

Figure 1: *VoiceGuard architecture. User U establishes a secure channel with the SGX enclave hosted at service provider P and sends sensitive voice data as well as user-specific adaptation data θ. Similarly, vendor V sends the sensitive models AM and LM through a secure channel. P securely processes U's voice data using V's models within an SGX Enclave.*

hypervisor. We assume that the adversary cannot perform invasive hardware attacks like extracting keys from the CPU. We also consider physical side-channel attacks, like differential power analysis [24], out of scope. We assume the enclave developer incorporated appropriate defense mechanism to protect against side-channel attacks leveraging micro-architectural effects [25, 26, 27].[1]

5. VoiceGuard Design

Our architecture VoiceGuard enables privacy-preserving and efficient speech processing on untrusted systems. VoiceGuard supports different deployment scenarios, i.e., the service provider is not necessarily a third party, but could also be the user or the vendor. Common to all scenarios is the basic setup, i.e., at least two *input parties* provide sensitive data while the computing platform is not trusted by at least one of them.

For the sake of simplicity we explain our solution based on the speech recognition scenario visualized in Fig. 1, where the service provider P is an untrusted third party, e.g., a cloud service provider. The vendor V's private input consists of speech recognition models. The user U's private input is the voice data. In this example, the output is sensitive as well and should only be made available to the user.[2]

VoiceGuard works in three phases: (I) preparation, (II) initialization, and (III) operation. In the first phase, user U and vendor V need to agree on the code to be executed in the enclave ("Encl. Code" in Fig. 1). In the second phase, the enclave code is instantiated. U and V use remote attestation (RA) to establish secure channels with the enclave through which they provision their respective encryption keys to the enclave. In the third phase, the enclave is ready to perform speech processing. Using the keys transmitted in the previous phase, U and V provide their respective inputs to the enclave in encrypted form. The result of the operation phase is encrypted with the user's key, so only she can decrypt it. Next, we describe the individual phases in detail:

Preparation Phase. First, U and V need to agree on the code to be run inside the SGX enclave. While SGX protects enclaves against accesses from the outside, they are nevertheless allowed to output data without any restriction. Therefore, U and V want to make sure that the enclave code only outputs non-sensitive data. The code typically comes from the vendor, i.e., V provisions the enclave code, ① in Fig. 1. Thus, V can easily ensure that no sensitive data will leave the enclave. The code itself is not necessarily confidential and is often open source. However, U has to carefully analyze the enclave code in a *vetting* process ② to verify that it does not contain functions which will leak her sensitive data. The vetting process could also be outsourced to a trusted third party, e.g., a government institution.

Additionally, the vendor provisions its acoustic model AM and language model LM to the service provider. Both are encrypted with the vendor's key K_V s.t. the service provider cannot access the vendor's intellectual property. At this stage, the models are not yet loaded inside an enclave, but are written to untrusted storage, e.g., the hard disk.

Initialization Phase. The enclave is created from the code provisioned by V earlier ③. The creation process is measured by the SGX-enabled CPU, i.e., a cryptographic hash of the initial memory content of the enclave is created and stored securely. If the enclave code is manipulated before or during the creation process, the measurement will produce a different result and the manipulation is detected. After the creation is finished, the code is isolated from all accesses and cannot be changed anymore.

The first operation performed by the enclave is the enclave initialization, during which the enclave generates a key pair for asymmetric cryptography operations like RSA [28],[3] with the public key PK shown in white in Fig. 1.

Next, U and V need to establish a secure channel with the enclave by provisioning their keys K_U and K_V, respectively, to the enclave. We will describe this process for U. The process for V is identical. VoiceGuard uses public key cryptography similar to Transport Layer Security (TLS) [29], which is widely used to secure web sites. The enclave sends its public key PK to U. However, U needs assurance that the received PK comes indeed from the correct enclave, i.e., the authenticity of PK must be established. This is done using the remote attestation (RA) feature of SGX, which generates a digital signature $\sigma(M, PK)$ that binds PK to the measurement M of the enclave, ④ in Fig. 1. In particular, the public key PK, which was generated *inside* the enclave, and the measurement of the initial enclave memory content are signed with the platform key. This signature can be verified using Intel's public key infrastructure (PKI) for SGX.

The user verifies the signature and checks that M matches her expectations, i.e., that the enclave has not been altered before or during creation. If both checks were successful, the user can be sure that PK belongs to the key pair generated by the correct enclave and that information encrypted with PK can only be decrypted inside that enclave. In step ⑤, U encrypts her key K_U with PK and sends the result $E_{PK}(K_U)$ to the enclave.

At the end of the initialization, the enclave shares a symmetric key with the user (K_U, the gray key in Fig. 1) and with the vendor (K_V, the black key in Fig. 1).

Operation Phase. The user sends encrypted inputs $E_{K_U}(input)$, i.e., audio samples, to the service provider. Since the input is encrypted with U's key, it can only be accessed by the enclave ⑥. If applicable, U also sends her user-specific adap-

[1] Our evaluation is performed without such protection mechanisms and thus does *not* reflect their impact on the performance results.

[2] The output could also be provided to one or multiple other parties.

[3] This process leverages the hardware random number generator of the CPU and can therefore not be influenced from outside the enclave.

tation parameters θ (e.g., i-vectors), which are also encrypted with K_U, to the enclave.

Inside the enclave, U's input is decrypted and passed to the speech recognition engine ("SR-Engine" in Fig. 1). The SR-Engine has two additional inputs, the acoustic model AM, typically a deep neural network (DNN), and the language model LM, typically a decoding graph. AM is provided by V and already stored encrypted at P. When AM is used, it is loaded into the enclave and decrypted using V's key K_V. Similarly, any adaptation parameters θ and the language model LM are loaded by the enclave, decrypted, and passed to the SR-Engine.

On-demand loading of AM or LM could leak sensitive information about their structure by observing access patterns. This can be prevented by storing this data in a randomized order, i.e., preventing an observer from learning useful information from observed access patterns [30].

The result of the speech processing is encrypted with K_U and sent back to the user ⑦.[4] Additionally, the SR-Engine may produce updated adaptation parameters θ, which are then encrypted with K_U and sent back to U.[5]

Once in the operation phase, the system can be queried repetitively by the user, thereby avoiding repeated preparation and initialization costs.

6. Evaluation

To show the effectiveness of VoiceGuard, we created a proof-of-concept implementation which embeds kaldi [3] in an SGX enclave using the Graphene library OS [31]. We ran experiments on two representative corpora: DARPA Resource Management (RM) [32] and Wall Street Journal (WSJ) [33]. Note that the purpose of these experiments is not to show improvements for certain training algorithms, but rather to prove that both regular and VoiceGuard decoding yield the exact same results with acceptable overhead. We chose RM and WSJ since they are well-established baseline recipes in kaldi which result in very different net and graph sizes for performance analysis.

For RM, we train on the speaker independent training and development set (about 4 000 utterances) and test on the six DARPA test runs: Mar and Oct'87, Feb and Oct'89, Feb'91, and Sep'92 (about 1 500 utterances in total), as a joint set. We use kaldi's `rm/s5` recipe and train the `nnet2_online` system with i-vectors. The resulting DNN is about 3 MB (9 hidden layers, 750 k parameters), the uni- and bigram decoding graphs are 0.5 MB and 2 MB, respectively. For details of the recipe, refer to kaldi at commit `cd6562`.

For WSJ, we train on the full SI284 set (about 60 h) and test on the Dec'93 development, Nov'92, and Nov'93 test sets. We use kaldi's `wsj/s5` recipe and also train the `nnet2_online` system with i-vectors. The resulting DNN is about 14 MB (15 hidden layers, 3.6 M parameters), the pruned trigram decoding graph is about 641 MB; since this is not about accuracy, no LM rescoring is applied. For details of the recipe, refer to kaldi at commit `ec98e7`.

In order to determine the overhead induced by VoiceGuard, we run kaldi on an Intel Core i7-7700 CPU @ 3.6 GHz over every corpus and report the run time of each test in Table 1. The overhead of VoiceGuard is between 39 % and 49 % for RM and between 98 % and 104 % for WSJ. The higher overhead for WSJ is due to its larger model (graph) size. In the current version of SGX, enclaves can only use up to 96 MB memory and rely on swapping to access additional data. Table 1 also shows the word error rate (WER) of each test, which is identical for VoiceGuard and baseline kaldi since they execute the same code on the same models resulting in identical lattices and transcriptions.

Table 1: *Performance of VoiceGuard w.r.t. baseline kaldi.*

Test	WER	Baseline (s)	VoiceGuard (s)	Overhead
RM-bigram	2.3 %	351	522	48.5 %
RM-unigram	15.4 %	585	815	39.3 %
WSJ (dev93)	18.1 %	1 427	2 854	100.1 %
WSJ (eval92)	13.4 %	876	1 736	98.2 %
WSJ (eval93)	15.5 %	518	1 058	104.3 %

We also differentiate between the time required to initialize the SR-Engine and to process a single file. The model setup time in the baseline is 0.04 s (RM-bigram) and 0.31 s (WSJ), while the setup time for the enclave and the models in VoiceGuard is 0.95 s (RM-bigram) and 23.55 s (WSJ). Note that this overhead is due to the initialization of enclave memory, occurs only once when the enclave is started, and is thus not repeated across multiple queries. The processing with RM-bigram of a 2.79 s audio file takes 0.32 s in the baseline and 0.50 s in VoiceGuard; with WSJ, the processing of a 6.12 s audio file takes 1.90 s in the baseline and 4.06 s in VoiceGuard. Thus, the overhead for the processing of one file is 56 % for RM-bigram and 114 % for WSJ, similarly to the overheads measured for the various batches, which indicates that the enclave setup overhead is amortized across multiple queries. Even though processing time is doubled in some cases, our results show that VoiceGuard enables privacy-preserving speech processing even in real time.

7. Conclusion

We proposed VoiceGuard, a novel architecture for privacy-preserving and efficient speech processing that supports user-specific models and can be deployed either in the cloud or on-premise. The evaluation of our prototype implementation demonstrates applicability for speech recognition in real time. Besides speech recognition, VoiceGuard's generic architecture works for related tasks such as speaker verification or voice biometrics, including emotion recognition and medical speech processing.

One core aspect to take into consideration when implementing this architecture in production systems is the model size: both AM and LM need to be loaded into the secure enclave, in turn causing computational overhead both at initialization and at run time. While small models such as RM (or models for speaker verification) require almost no memory, typical high-accuracy ASR systems would use much larger models than the WSJ models evaluated in this experiment.

Thus, as part of future work, we will explore distributing the processing across multiple SGX-enabled nodes and optimize performance for more accurate models with larger memory requirements. We will also determine the overhead incurred by employing protection mechanisms against side-channel attacks.

8. Acknowledgments

This work was co-funded by the DFG as part of projects P3, S2, and E4 within CROSSING, by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

[4] The result could also be sent to a different party, even a third party.

[5] U can decrypt θ and re-encrypt it to make individual requests from the same user unlinkable.

9. References

[1] M. Marlinspike, "Technology preview: Private contact discovery for Signal," https://signal.org/blog/private-contact-discovery/, September 2017.

[2] M. Russinovich, "Introducing Azure confidential computing," https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, September 2017.

[3] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, "The Kaldi Speech Recognition Toolkit," in *Workshop on Automatic Speech Recognition and Understanding (ASRU)*. IEEE Signal Processing Society, 2011.

[4] P. Mohassel and Y. Zhang, "SecureML: A System for Scalable Privacy-Preserving Machine Learning," in *Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[5] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "DeepSecure: Scalable Provably-Secure Deep Learning," *CoRR*, vol. abs/1705.08963, 2017.

[6] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious Neural Network Predictions via MiniONN transformations," in *Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[7] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications," in *Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 2018, to appear.

[8] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "EzPC: Programmable, Efficient, and Scalable Secure Two-Party Computation," *IACR Cryptology ePrint Archive*, vol. 2017/1109, 2017.

[9] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A Low Latency Framework for Secure Neural Network Inference," *IACR Cryptology ePrint Archive*, vol. 2018/073, 2018.

[10] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *International Conference on Machine Learning (ICML)*. JMLR, 2016.

[11] E. Hesamifard, H. Takabi, and M. Ghasemi, "CryptoDL: Deep Neural Networks over Encrypted Data," *CoRR*, vol. abs/1711.05189, 2017.

[12] P. Koeberl, V. Phegade, A. Rajan, T. Schneider, S. Schulz, and M. Zhdanova, "Time to Rethink: Trust Brokerage Using Trusted Execution Environments," in *Trust and Trustworthy Computing (TRUST)*, ser. LNCS, vol. 9229. Springer, 2015.

[13] K. A. Küçük, A. Paverd, A. Martin, N. Asokan, A. Simpson, and R. Ankele, "Exploring the Use of Intel SGX for Secure Many-Party Applications," in *System Software for Trusted Execution (SysTEX)*. ACM, 2016.

[14] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi, "Secure Multiparty Computation from SGX," in *Financial Cryptography and Data Security (FC)*, ser. LNCS, vol. 10322. Springer, 2017.

[15] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *USENIX Security Symposium*. USENIX, 2016.

[16] S. Chandra, V. Karande, Z. Lin, L. Khan, M. Kantarcioglu, and B. Thuraisingham, "Securing Data Analytics on SGX with Randomization," in *European Symposium on Research in Computer Security (ESORICS)*, ser. LNCS, vol. 10492. Springer, 2017.

[17] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving Machine Learning as a Service," *CoRR*, vol. abs/1803.05961, 2018.

[18] M. A. Pathak, B. Raj, S. Rane, and P. Smaragdis, "Privacy-Preserving Speech Processing: Cryptographic and String-Matching Frameworks Show Promise," *IEEE Signal Processing Magazine*, vol. 30, no. 2, 2013.

[19] C. Glackin, G. Chollet, N. Dugan, N. Cannings, J. Wall, S. Tahir, I. G. Ray, and M. Rajarajan, "Privacy preserving encrypted phonetic search of speech data," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017.

[20] Intel, "Intel Software Guard Extensions Programming Reference," 2014. [Online]. Available: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf

[21] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[22] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[23] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[24] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology (CRYPTO)*. Springer, 1999.

[25] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *Network & Distributed System Security Symposium (NDSS)*. Internet Society, 2017.

[26] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu," in *Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 2017.

[27] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A. Sadeghi, "DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization," *CoRR*, vol. abs/1709.09917, 2017.

[28] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Communications of the ACM (CACM)*, vol. 21, no. 2, 1978.

[29] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," Internet Requests for Comments, RFC 5246, 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5246.txt

[30] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "HardIDX: Practical and Secure Index with SGX," in *Conference on Data and Applications Security and Privacy (DBSec)*, ser. LNCS, vol. 10359. Springer, 2017.

[31] C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *USENIX Annual Technical Conference (USENIX ATC)*. USENIX, 2017.

[32] P. Price, W. Fisher, J. Bernstein, and D. Pallett, "Resource Management RM1 2.0," Linguistic Data Consortium, Philadelphia, USA, 1993.

[33] J. Garofalo, D. Graff, D. Paul, and D. Pallett, "CSR-I,II (WSJ0,1) Complete," Linguistic Data Consortium, Philadelphia, USA, 2007.

DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization

Ferdinand Brasser
Technische Universität Darmstadt
ferdinand.brasser@trust.tu-darmstadt.de

Srdjan Capkun
ETH Zurich
srdjan.capkun@inf.ethz.ch

Alexandra Dmitrienko
University of Würzburg
alexandra.dmitrienko@uni-wuerzburg.de

Tommaso Frassetto
Technische Universität Darmstadt
tommaso.frassetto@trust.tu-darmstadt.de

Kari Kostiainen
ETH Zurich
kari.kostiainen@inf.ethz.ch

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Recent research has demonstrated that Intel's SGX is vulnerable to software-based side-channel attacks. In a common attack, the adversary monitors CPU caches to infer secret-dependent data accesses patterns. Known defenses have major limitations, as they require either error-prone developer assistance, incur extremely high runtime overhead, or prevent only specific attacks.

In this paper, we propose data location randomization as a novel defense against side-channel attacks that target data access patterns. Our goal is to break the link between the memory observations by the adversary and the actual data accesses by the victim. We design and implement a compiler-based tool called DR.SGX that instruments the enclave code, permuting data locations at fine granularity. To prevent correlation of repeated memory accesses we periodically re-randomize all enclave data. Our solution requires no developer assistance and strikes the balance between side-channel protection and performance based on an adjustable security parameter.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Trusted computing**.

KEYWORDS

SGX; side channel defense; data randomization

ACM Reference Format:
Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. 2019. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359789.3359809

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359809

1 INTRODUCTION

Intel Software Guard Extensions (SGX) [18, 35] enable execution of security-critical application code, called *enclaves*, in isolation from the untrusted system software. SGX was designed to ensure confidentiality of enclave data and integrity of enclave execution and is used in a number of academic works [4, 9, 14, 15, 20, 26, 41, 58, 63, 70].

Recent research has, however, demonstrated that SGX isolation can be violated using software-based side-channel attacks. In SGX, memory management, including paging, is left to the untrusted OS [18]. By monitoring page usage, the OS can learn coarse-grained enclave control flow or data access patterns [72, 77]. Enclave data can also be inferred by monitoring CPU caches that are shared between the enclave and the untrusted software, enabling more fine-grained information leakage [11, 31, 32, 51, 64]. Such attacks can defeat one of the main benefits of SGX—the ability to compute over private data on an untrusted (cloud) platform.

The problem of side-channel leakage has been studied extensively. Oblivious RAM (ORAM) [69] and Oblivious Execution [44, 45, 48] are well-known defensive techniques. Obfuscuro [1] implements those techniques for SGX enclaves, hiding all access patterns. The main drawback is an extremely high runtime overhead (83× on average and up to 220×). Another common defense is manual code hardening that is typically used by developers of cryptographic algorithms to make their implementations side-channel resilient [12]. This defense is not easily applicable to enclaves written by developers who are not security experts. Recent research has also proposed SGX-specific defenses. T-SGX [66] and Déjá Vu [17] use the processor's transactional memory features to prevent attacks that interrupt the victim enclave repeatedly. Such features are available only in a subset of SGX processors and the defense only protects against attacks that leverage interrupts. Cloak [33] and Raccoon [59] hide memory accesses to developer-annotated enclave data, but relying on the developer to mark all (possibly non-obvious) secret data correctly can be very error-prone. In summary, all known defenses either impose extremely high runtime overhead, rely on the developer, require functionality that is not available in all CPUs, or mitigate only specific side channels.

Our goals and approach. In this paper we focus on information leakage caused by *data access* monitoring. Our goal is to provide an *automated* tool that provides side-channel protection without

developer assistance and enables an *adjustable* trade-off between security and performance.

We focus on data accesses, as they are the target of many recent SGX attacks [11, 31, 51, 64]. Preventing control flow leakage is also important, but an orthogonal problem to our work. We build an automated tool because, similar to the development of other software, not all enclave developers are security experts and many would fail to correctly use solutions that require identification of potentially subtle sources of leakage for manual annotation. Instead, our primary goal is to strike the balance between provided protection and performance. While our tool can be configured to prevent all the leakage, this would incur a prohibitive performance penalty for most applications. Instead, we aim to give a means to enclave developers to get the best possible protection for a given application and performance overhead.

The main idea of our approach is to randomize all data locations in the enclave's memory at fine granularity. The enclave generates a secret randomization key and based on that computes a permutation for every memory address. As a result, the adversary cannot map the observed (permuted) memory address to the actual address, regardless of the channel he uses to make observations [11, 31, 32, 51, 64, 72, 73, 77]. Because all data is randomized without the need to understand its structure or semantics, we call our approach *semantic-agnostic data randomization.*

Randomization is a well-known hardening technique, but our approach is different from the existing solutions that randomize code by leveraging its known structure, such as functions or blocks. Due to the well-known difficulty of C/C++ code analysis and pointer tracking, no similar structure is available for data [6]. Indeed, existing randomization tools like SGX-Shield [65] focus on randomizing the code and do not tackle the problem of data randomization. Thus, they cannot prevent attacks that exploit data accesses, such as [11, 31, 51, 64].

Challenges and results. Secure and practical realization of our approach imposes a number of technical challenges. The first challenge is secure and efficient permutation computation under adversarial monitoring. If the adversary is able to derive information from the process of address permutation, he can revert the randomization. The second challenge is efficiency — computing a permutation for every data access is expensive and causes a high overhead. The third problem is information leakage through repeated memory accesses. Although an individual access is effectively hidden from the adversary, repetitive access patterns may allow (permuted) address correlation and leakage, i.e., correlation attacks.

In this paper, we tackle the above mentioned challenges and design and implement a compiler-based tool called DR.SGX (***D**ata Location **R**andomization for **SGX***) that instruments enclave code at compile time such that all memory locations used to store enclave data (in the heap) are permuted at cache-line granularity during run time. We realize the permutation securely using small-domain encryption [5] and leveraging the CPU's hardware acceleration units (AES-NI). To address correlation attacks, our tool allows periodic re-randomization of enclave data: more aggressive re-randomization rates hide repeated memory access patterns better at the cost of higher run-time overhead.

The basic runtime overhead of DR.SGX is 4.36× without re-randomization. Using different re-randomization rates, we measured an overhead approximately between 5× and 11×. We acknowledge that this is a significant performance penalty, but emphasize that our solution is at least one order of magnitude faster than complete ORAM schemes like Obfuscuro [1]. Additionally, we note that this overhead only applies to the SGX enclave, which handles just the security-critical part of an application.

Our security evaluation reveals that the protection provided by DR.SGX depends on the target enclave. Enclaves where predictable data access patterns, like initialization routines, are soon followed by secret-dependent data accesses, require aggressive re-randomization to prevent leakage, incurring higher overhead. In a corner case, our solution can prevent any leakage by re-randomizing enclave memory after every memory access, effectively functioning as an ORAM implementation. However, enclaves where secret-dependent accesses do not happen (soon) after predictable accesses can be strongly protected with much lower overhead.

Contributions. This paper makes the following main contributions:

- *Novel approach.* We propose a novel approach called semantic-agnostic data randomization as a defense against side-channel attacks on SGX.
- *New tool.* We design and implement a tool called DR.SGX that instruments code to permute an enclave's data memory locations at cache-line granularity and re-randomize them repeatedly.
- *Evaluation.* We evaluate the performance of our system, analyze possible leakage, and show how previous attack targets can be protected.

The paper is organized as follows: Section 2 defines our problem. Section 3 presents our approach and Section 4 details on our implementation. We evaluate DR.SGX's performance in Section 5 and analyze its security in Section 6. Section 7 reviews related work, Section 8 provides discussion and Section 9 concludes the paper.

2 PROBLEM STATEMENT

In this work we focus on systems that provide an isolated execution environment that is implemented as an execution mode of the main CPU. In particular, the CPU's shared resources, like caches, are used by all execution modes of the CPU and thus are shared between isolation domains. Our work is targeted towards Intel SGX, however, the same model also applies to other architectures like ARM TrustZone [2] and SANCTUARY [10] or software-based isolation solutions [49].

Problem space. Side-channel attacks on software in general, and SGX in particular, come in many different forms. Any kind of resource use that is influenced by the software's execution and can be observed by the adversary can serve as a side channel. For instance, the use of electricity as well as effects thereof like electro-magnetic emission, or the use of shared CPU caches. In this work we focus on *software* side channels, i.e., such that are observable by a software program running on the target machine, precluding physical or hardware side-channel attacks.

In the realm of software side-channel attacks a number of distinct variants exist. On one hand, different shared resources can be used as a side channel, like the different caches of the CPU, or the virtual memory management. On the other hand, side-channel attacks can target different information, including sensitive access patterns to data as well as secret dependent code execution paths.

In this work we focus on software attacks that target *data accesses* and consider attacks aiming to infer the control flow of a program as an orthogonal problem. Our rationale is two-fold. First, many side-channel attacks on SGX have been based on data access patterns [11, 31, 51, 64]. Furthermore, our solution can be combined with protections against control flow leakage attacks, for example with the Zigzagger approach proposed by Lee et al. [42].

Adversary model. The adversary's goal is to extract sensitive information from an isolated execution environment (enclave) [3, 8] through cache side-channel attacks (including CPU-internal caches like the translation look-aside buffer [32]) and/or paging side-channel attack [72, 77]. Sensitive data in this context are not limited to cryptographic keys, which are the "classical" targets of side-channel attacks. Instead, sensitive data have to be seen much broader, for instance, when processing privacy-sensitive data in the cloud [11].

The adversary can freely configure and modify all software of the system, including privileged software like the operating system (OS). He knows the initial memory layout of the enclave, i.e., the code and initial data of the enclave. Furthermore, we assume that the adversary can initiate the enclave arbitrarily often.

However, the adversary cannot directly access the memory of the enclave. The internal processor state (e.g., the CPU registers) is inaccessible to the adversary, in the event of an interrupt the state is securely stored in an isolated memory region. The adversary cannot modify the code or initial data of the enclave, as enclave's integrity can be verified using remote attestation.

We consider our work orthogonal to the recently discovered platform vulnerabilities Meltdown [43] and Spectre [39] that leverage transient execution to read secrets across isolation boundaries. Although these vulnerabilities apply to SGX enclaves as well [16, 71], Intel has already issued security updates for SGX that address such attacks [16]. Also, SGX platform keys from unpatched (and thus potentially compromised) platforms can be identified at the time of attestation and revoked [16]. The more general problem of data-access driven side-channels is much harder to solve in architectures like SGX. DR.SGX addresses this latter and more difficult problem.

We assume the position of the attacker to be as strong as possible and therefore we will assume him to have a noise-free cache side-channel and to be able to obtain a "perfect cache trace" of the enclave. This means that he can observe all memory accesses of an enclave, e.g., using a cache attack technique such as Prime+Probe [54]. He can precisely determine which cache line has been used by the enclave and also the order in which the cache lines have been accessed. The adversary cannot extract information which is more fine grained than accesses to cache lines, i.e., the offset inside a cache line is not observable to him (see Section 8 for a discussion of possible attacks with finer granularity). Additionally, for each memory access, the adversary can gain information about the accessed memory pages of an enclave [72, 77].

More formally, trace $t = \{c_1, p_1\}, ..., \{c_n, p_n\}$ is an ordered list of side-channel observation pairs that capture every memory access that the victim enclave makes. In each observation pair, c_i is the part of the memory address that determines the cache line the accessed address gets mapped to and p_i is the part of the address that determines the accessed memory page. On current Intel CPUs the cache line size is 64 bytes, thus, the last six bits of an address are oblivious to the adversary.

Design goals. General statements about which memory accesses of a program could leak information are hard to make in practice. All memory accesses must be assumed to potentially leak information if the attacker can associate them with relevant data elements or structures. For the adversary it is sufficient to distinguish two memory locations to learn one bit of information. Those memory locations could be two different data structures, e.g., two variables, or different elements within the same data structure, e.g., different entries in a table. To protect all possible programs, the data structures of a program and the elements within data structures both need to be randomized.

The goal of our work is to provide a protection mechanism against side-channel attacks that can be applied to *arbitrary enclave programs without developer assistance.* In particular, the developer must not be required to follow any rules or guidelines for programming his application or add annotations to the source code. While annotating "critical" data in general helps improving the performance of most solutions, it is also very error-prone: especially in non-cryptographic applications, it is not always obvious which accesses to data objects might leak sensitive information. This is crucial as most software developers are not security experts and cannot comprehensively identify data that could leak information.

The goal of DR.SGX is to provide a trade-off between security and cost in the design space reaching from unprotected processes, over plain SGX enclaves, enclaves with DR.SGX to oblivious RAM (ORAM) solutions. On the one hand, plain SGX enclaves provide basic data protection with little performance penalty; on the other hand, schemes like Obfuscuro [1] that implement ORAM for every memory access impose very high performance overheads (83× on average and up to 220×). DR.SGX strives to protect enclaves better than plain SGX while keeping the performance overhead at least one order of magnitude lower than systems like Obfuscuro. DR.SGX's security parameter (the re-randomization window w: see Section 3) allows it to be configured to cover the spectrum between plain SGX and full ORAM for data accesses. With $w = 1$ DR.SGX implements ORAM, admittedly in a costly way. On the other hand, $w = \infty$ only randomizes the initial memory layout of an enclave, which can be sufficient for some enclaves; we discuss this scenario in Section 6. For most enclaves a window size between those two extremes can be chosen. We evaluate different windows sizes in Section 5.

3 DR.SGX

Our core idea is to break the link between side-channel observations made by an attacker and the sensitive information processed by the victim. Side-channel attacks inherently rely on the correlation between an observable effect and the data the attacker aims to extract. Our defense obfuscates the link between memory locations and

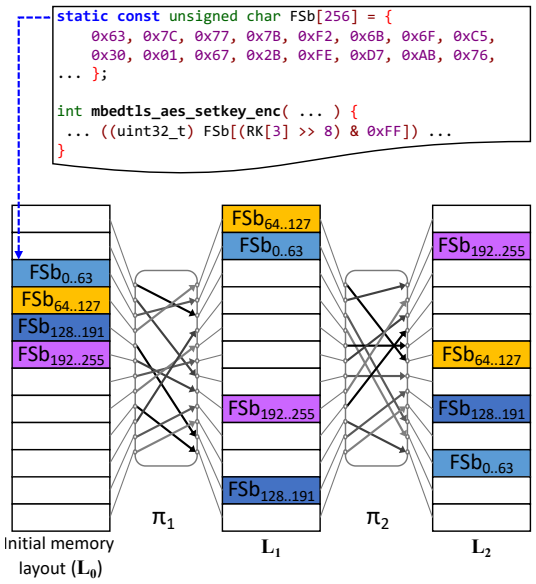

Figure 1: DR.SGX's memory block randomization splits large memory structures like arrays into small blocks and reorders them. During the run time of an enclave its memory layout are re-randomized using the permutation function π. Each memory block is the size of a cache line (64 B), i.e., the finest granularity observable by the adversary.

data elements. Data elements are located at randomized memory locations, so the adversary cannot deduce which data element was accessed from an observed memory access location. The adversary no longer learns *which* data element was accessed but only learns that *some* data element was accessed.

DR.SGX splits enclave memory into small blocks that are randomly reordered, resulting in an unpredictable memory layout from the adversary's point of view. Figure 1 illustrates the concept on the example of the S-box of an AES implementation. By default the S-box (FSb) is stored as an array in consecutive memory at a predictable location, shown on the left as initial memory layout L_0 in Figure 1. Through a cache side channel an adversary can observe which part of the S-box is accessed. Since the accesses to the S-box depend on the secret key the adversary can use this information to recover the key. However, the adversary cannot observe accesses to individual bytes of the S-box but only at the granularity of cache lines (64 bytes). DR.SGX divides *all* data memory of an enclave into blocks of cache line size, illustrated by the blocks forming L_0 in Figure 1. These blocks are reordered by a permutation function π_1, resulting in a randomized memory layout L_1. Throughout the runtime of an enclave the memory layout is constantly re-randomized, by applying a permutation function π_2 on L_1 a new and different memory layout L_2 is created. As a result, the memory locations and thus the cache lines corresponding to the S-box are frequently changing, hindering the adversary's ability to link observed (cache or paging) accesses to the S-box.

3.1 Requirements and Challenges

Below we describe the main challenges to tackle when implementing this idea.

Semantic gap. Providing side-channel protection through data randomization without developer assistance (e.g., code annotations) is a challenging task due to the semantic gap that is inherent to unsafe languages like C and C++. Currently C and C++ are the only programming languages officially supported in the software development kit (SDK) that Intel provides for the development of SGX enclaves.

Re-randomization. Randomizing the memory layout of a program once to prevent an adversary from learning which data has been accessed is not sufficient. The adversary can determine the relation of memory locations and data objects based on various information. For instance, the initialization of data structures can reveal data locations. In the example in Figure 1, the S-box is initialized during the creation of the enclave, however, other AES implementations initialize the S-box at run time which allows the adversary to learn the locations of all parts of the S-box array *after* the initial randomization of the memory layout. Similarly, access frequency can reveal the randomized location of data elements: if a particular object is accessed a predictable number of times the adversary can identify the object by finding the memory location that was accessed the expected numbers of times (frequency analysis). To thwart the adversary in recovering the randomized memory location of data objects, their locations need to be changed throughout the runtime, such that the adversary cannot link data accesses to data objects.

(Re-)randomization under attacker's observation. All memory-related actions of the attacked enclave can be observed by the adversary, including those required during the initial data randomization and during the re-randomization of the memory layout. The initial (un-randomized) memory layout is known to the adversary, i.e., he can monitor memory events while data is copied to its randomized locations. Similarly, if the adversary managed to recover information about the randomized memory layout L_n the adversary could link the re-randomization operations used to transfer data from L_n to L_{n+1} and thus also gain knowledge about the new layout L_{n+1}. Therefore, the randomization has to be done in such a way that its effects are not observable by the adversary.

3.2 DR.SGX Design

Our solution, a compiler-based tool called DR.SGX, addresses the design goals and challenges described above by randomizing *all* program data at fine granularity and re-randomizing the data continuously throughout the run time of the program.

Figure 2 shows the system view of DR.SGX. The trusted computing base (TCB) of an SGX enclave includes the CPU package and an isolated section of the main memory (RAM). However, the CPU caches, translation look-aside buffer (TLB) and the page tables are observable by the adversary. The data cache of the CPU can be used to observe memory access patterns of an enclave. On the other hand, the paging mechanism can be exploited in different ways to learn about memory reads and writes by an enclave. By observing cache conflicts in the TLB, the adversary learns which memory

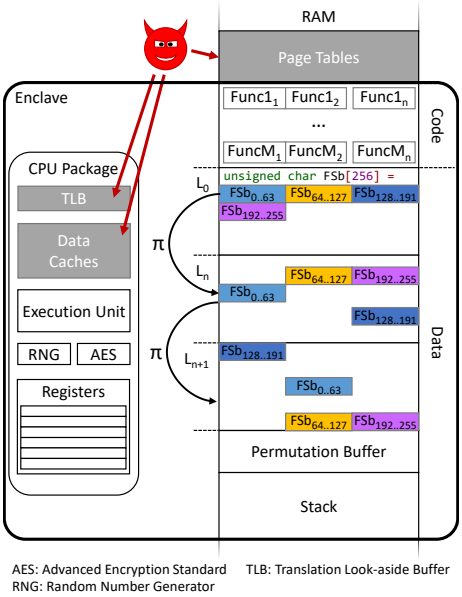

Figure 2: DR.SGX's system design. The main memory of an enclave is not directly accessible by the adversary, however, the adversary can observe memory access indirectly through cache and paging side channels. The CPU's internal state stored in registers and/or special function units (e.g., the AES engine) are not observable by the adversary.

pages were used. Additionally, the adversary has control over the page tables also allowing him to learn which memory pages an enclave accessed.

However, an SGX enclave also includes components that cannot be attacked through a software side channel. The CPU's registers and accesses to them cannot be observed by the adversary.[1] Also the execution unit and special function units, like the random number generator (RNG) or the AES engine, are secure when operating over registers. DR.SGX combines these parts and function units of SGX that are secure against side channels, to obfuscate main memory accesses to the adversary.

DR.SGX performs randomization at granularity of cache lines, the finest granularity at which the adversary can distinguish memory accesses (Section 2). Figure 2 shows how DR.SGX uses a random permutation function π to reorder the program's data in memory. Since the adversary cannot identify individual elements within a single cache line, accesses to the first array element (`FSb[0]`) and the 64th element (`FSb[63]`) are indistinguishable for the adversary. The randomization is based on secret values which are generated and only accessible *inside* the enclave and only processed by the hardware AES engine of the CPU. The CPU's AES engine holds all state and intermediate results in registers which are not observable by the adversary, hence, the adversary cannot learn about π through cache or paging side channels.

[1]The LazyFP [68] attack cannot be used on SGX enclaves, since the register state is cleaned by the processor before exiting the enclave.

DR.SGX randomizes global variables and the heap. The stack cannot be easily randomized, since the hardware expects it to be contiguous. Thus, variables on the stack larger than a cache line are moved to the heap, and replaced by a pointer on the stack. The remaining variables are protected using multiple memory layouts: for every function n variants are created (Func1$_1$, Func1$_2$, ..., FuncM$_n$ in Figure 2), all with different stack memory layouts. On every invocation of a function one of its n variants is chosen randomly.

The size of the memory region (heap) for the enclave's data is a parameter of the permutation function π (see Section 4).

Memory access instrumentation. DR.SGX performs randomization on cache line granularity for two reasons: (a) randomizing at finer granularity provides no security advantages, and (b) randomizing in a data structures aware fashion is impractical due to the semantic gap. Our randomization requires that *all* memory accesses are instrumented, which we ensure using a compiler pass. The program code determines the memory location (i.e., address) of the data in the original, un-randomized layout. Then, before the access is performed, the randomized location of that address is calculated. The data is then accessed in its new, randomized location.

As we will elaborate in later sections, the cost of performing the randomization calculation for *every* memory access is significant. We overcome this problem by implementing a "permutation buffer". The permutation buffer, similar to an address translation cache, holds the randomized locations of recently used data. Hence, for data locations stored in the permutation buffer the function π does not need to be recalculated. However, accesses to the permutation buffer itself must be protected from leaking information. Therefore the buffer is accessed in an oblivious way.

Initial randomization. The initial randomization of the enclave's data needs to be done in a way that cannot be observed by the adversary, to keep him from learning the randomization function π or the new memory layout. In particular, if the adversary can observe a read operation from the un-randomized initial memory layout and a subsequent write operation to a randomized address, he can link data structures to the randomized memory locations.

A general approach to break this linkage is to load a set of data into CPU registers (register operations cannot be tracked by the adversary) and write the data in a random fashion to their new locations. This approach, however, is limited in the amount of the data that can be loaded at once into registers, enabling the adversary to learn partial information about the randomized memory layout.

DR.SGX uses a randomization method which hides fine-grained (cache-line granularity) memory locations from the adversary. Specifically, we use *non-temporal writes* [36] that evade the CPU's caches, therefore the adversary cannot observe memory addresses written during the initial randomization. Although the non-temporal writes prevent accesses to the new memory layout L_1 from being cached, the adversary can still observe the written memory locations through the more coarse-grained paging side-channel (that is, the adversary's trace contains a page event p_i, but no cache event c_i for the non-temporal write). This allows him to know, for each memory block read from the previous memory layout L_0, to which memory page it was written in L_1. However, multiple cache lines are written to each page: assuming 4 KB pages, 64 cache-line-sized memory blocks will be written to the same page.

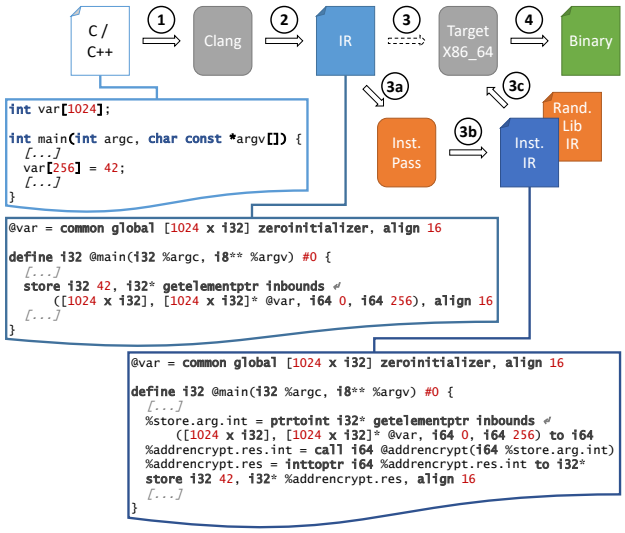

Figure 3: Code instrumentation with DR.SGX. Before each memory access the randomized memory address is calculated. The calculation is done by a function provided by the DR.SGX library (Rand. lib), which can be written in C / C++ and is included in the instrumented binary. The snippets show the instrumentation of a sample store instruction.

To hide this access pattern the initial randomization of DR.SGX accesses *all* memory pages of L_1 for each memory block that is moved, see Section 6.2.

DR.SGX continuously re-randomizes the memory layout. Starting from the initial memory layout L_0 a random permutation function π_1 is applied to derive the first randomized layout $L_1 = \pi_1(L_0)$. After a configurable window w the memory layout is re-randomized, applying π_2 to derive $L_2 = \pi_2(L_1)$.

Like with the initial randomization, the adversary (who can observe reads from L_n and writes to L_{n+1}) could link those operations to learn the relation between those memory layouts. Again, DR.SGX uses non-temporal writes to hide this information. In Section 6 we explain how a small number of re-randomization rounds hides the location of the element from the adversary completely.

4 DR.SGX IMPLEMENTATION

This section provides further details of DR.SGX. We explain how we implemented the key-components of DR.SGX: access instrumentation, permutation computation, initial randomization, permutation buffering, and re-randomization. Throughout this section we will refer to *data* memory regions or *data* memory accesses simply as memory regions and accesses (omitting *data*).

4.1 Memory Access Instrumentation

DR.SGX randomizes the memory locations of an SGX enclave's data. The enclave, however, has been developed targeting a linear (virtual) memory model. Therefore, each memory access of an enclave has to be instrumented to determine the correct randomized memory location of the data element that is meant to be accessed.

We extended the LLVM compiler [47] to instrument the enclave code, working at the intermediate representation (IR) level. Figure 3 shows on the top the high-level compile process of LLVM. A source file on the left is translated by the compiler front-end ①, Clang in the case of C/C++, into a LLVM intermediate representation (IR) ②. The IR is then translated by the back-end ③ into target architecture specific binary code ④, which in our case is Intel x86 64-bit. With DR.SGX the IR file is processed by a compiler pass ③a that instruments all memory access instructions (instrumentation pass) before it is translated into machine code ③c. Furthermore, DR.SGX adds a small library ③b, which contains functions used to perform the randomization. This library can be written in a high-level language like C/C++ and is translated into IR as well.

Additionally, the instrumentation pass examines all allocations on the stack and transforms those which are larger than a single cache line into heap allocations. A pointer to the heap allocation is placed on the stack and the code is modified to access the heap allocation instead of accessing the stack.

Instrumentation example. Figure 3 illustrates the instrumentation of a write access to an array. The code snippet in the C file shows a write access to the 257-th element of an integer array `var`. The code snippet in the middle shows the intermediate representation (IR) of the write operation. The array is accessed by calculating the pointer to the 257-th element of the array, using the LLVM function `getelementptr`. The value 42 is then stored into this memory location. The instrumented IR is shown in the bottom code snippet. Again, a pointer to the 257-th element of `var` obtained using `getelementptr` and stored in the variable `store.arg.int`. However, before storing the value 42, `store.arg.int` is passed to the permutation function `addrencrypt`. The function returns the permuted location of the 257-th element of `var`, which gets cast from an integer value to a pointer value (`inttoptr`). The value 42 is then stored to the permuted location `addrencrypt.res`.

4.2 Random Permutation

DR.SGX uses run-time data randomization, which is required for both the unobservable initial randomization as well as the re-randomizations. This means that the randomized location of data must be recovered dynamically. Using a purely random permutation would require storing extensive meta-data, which would then need to be accessed in an unobservable way.[2] Therefore, DR.SGX uses a pseudo-random permutation function to determine the random location of data. This approach has two advantages: (1) collisions, i.e., different element mapped to the same location, are inherently avoided, and (2) randomized locations can be computed based on a non-secret algorithm and a key, which is *small* compared to the meta-data in the naive approach. However, the permutation function itself must be resilient against side-channel attacks, otherwise the adversary can learn the randomization secret and disclose the accessed memory locations.

We use small-domain encryption for our random permutation function. The domain size must be in the order of memory size used by the enclave employing DR.SGX (divided by the size of a

[2]The need to maintain meta-data is one of the main problems when using ORAM to protect SGX enclaves from side-channel attacks targeting the enclave's main memory accesses.

cache line). In particular, we use the FFX Format-Preserving Encryption scheme, which is based on a 10-round Feistel network [5]. As the underlying block cipher for FFX we used AES, for which the hardware acceleration extension AES-NI [36] is available in all SGX-enabled CPUs. AES-NI provides both good performance and resiliency against cache-based side-channel attacks.

Our implementation only supports single-threaded enclaves. However, standard software-engineering techniques can be employed to extend the support to multi-threaded enclaves. Only the re-randomization operations need to be synchronized between threads.

4.3 Initial Randomization

The initial randomization is particularly challenging since the adversary knows the initial memory layout of an enclave. If we used standard write operations to copy data from the initial data section L_0 to the randomized section L_1, the adversary would be able to learn the randomized layout.

In DR.SGX we use non-temporal write instructions to tackle this problem [36]. Non-temporal write instructions provide the processor with the meta-information that the data will not be used again soon by the program and it is not necessary to store them in the cache. On current Intel processors memory write operations using this instruction immediately affect the DRAM and are not buffered in the CPU's cache,[3] i.e., they are invisible to the adversary. Page-granularity side-channels information is hidden by accessing all heap memory pages for each block.

The secret keys we need as input to our random permutation are generated by the hardware random number generator *inside* the enclave. We use `rdseed` to obtain true random numbers from the CPU [36]. This way the adversary cannot influence or obtain the secret key.

4.4 Stack Randomization

DR.SGX uses the stack only for data elements that are smaller than a cache line, all other data are moved to the heap where they are subject to (re-)randomization. For the remaining data elements on the stack we use an approach inspired by the code randomization method introduced by Crane et al. [19]. The stack layout of each function is randomized by reordering the local variables on the stack. At compile time n variants of each function with different stack layouts are generated. At run time one function variant is chosen at random every time it is invoked. DR.SGX uses $n = 10$ variants for each function, as the empirical evaluation [19] suggests.

4.5 Permutation Buffer

Performing the calculation for the pseudo-random permutations is costly and needs to be performed for each memory access. To improve the performance we introduced a buffer for memory translations (Permutation Buffer in Figure 2). Permutation is performed at cache line granularity, i.e., all bytes in one cache line in L_0 are mapped as a single block. When this block is moved to L_1 it will, with high probability, be mapping to a different cache line, and to yet another cache line in L_2, and so on. On recent x86 processors a cache line is 64 bytes, thus, by storing the result no extra calculations are necessary for memory accesses that fall within the same cache line. Our buffer is currently 1 KB which allows for a direct-mapped storage of permutation results for 256 translations. To prevent leakage through our permutation buffer we access it in a way which is oblivious to the adversary. For each read operation to the buffer we simply access *all* CPU cache lines in our permutation buffer. Moreover, we randomize the location of the items in the permutation buffer by performing an xor operation with a randomly-generated value before determining which buffer item to use. The random value changes and the buffer is invalidated every time a re-randomization happens.

4.6 Re-Randomization

DR.SGX constantly re-randomizes the memory layout of an enclave. Figure 2 shows the overall memory layout. The blocks are copied from L_n to L_{n+1} in the same order as they appear in L_n, so the adversary only observes reads to every block in L_n, in order. Like in the initial permutation, non-temporal write operations are used to hide fine-grained writes.

For each cache-line-sized memory block in L_n, DR.SGX needs to compute the corresponding addresses in L_n and in L_{n+1}. Hence, the cost of re-randomization primarily comes from the permutation calculations required. However, the pipelining of AES instructions in the CPU makes encrypting multiple addresses together faster than encrypting them sequentially. This reduces the cost for the re-randomization and leads to better overall performance of DR.SGX.

5 PERFORMANCE EVALUATION

We evaluated the performance of DR.SGX using the benchmark suite Nbench [13].[4] We use Nbench because it has been previously used to analyze SGX performance [65], it relies only marginally on the file system, and it is relatively simple (5217 LoC), so it can easily be adapted to run inside an SGX enclave. The original version relies on timestamps to run each benchmark for an equal amount of time; since timestamps are not available in SGX enclaves we manually chose for each benchmark the lowest number of iterations that yielded a run time greater than 100 ms. We measured the run time of the benchmarks by briefly switching to the non-SGX mode and reading the hardware time stamp counter. We measured the overhead due to this mode switch and it is negligible compared to the overall run time. Our test system is equipped with an Intel Skylake i7-6700 processor clocked at 3.40 GHz, 128 MB Enclave Page Cache, running Ubuntu 14.04.4.

Memory overhead. The memory overhead of DR.SGX is mainly due to (1) heap randomization and (2) stack randomization. For the heap randomization two memory areas as large as the heap need to be reserved while the re-randomization is in progress. In our evaluations the heap size was set to values between 512 KB and 4 MB. Whenever the re-randomization is ongoing an additional 100% for the heap size is required. Stack randomization is based on providing n variants for each function. This increases the memory required

[3] We verified this behavior on a Skylake test system by issuing a non-temporal write followed by a read from the same cache line, and verifying that the read generates a cache miss on all three cache levels.

[4] Benchmarking SGX code can be challenging, since well-known benchmark suites rely on a number of features, including system calls, timestamps, and the file system, which are not directly available in SGX.

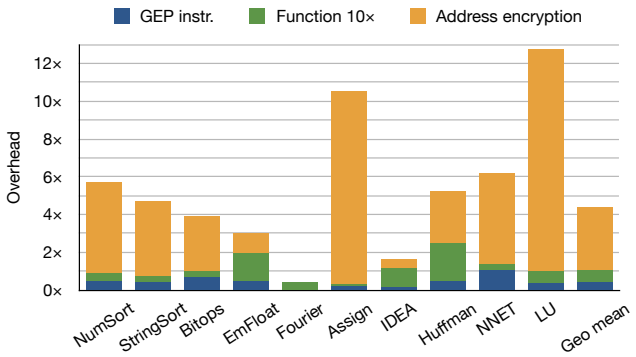

Figure 4: Overhead of each benchmark, using various subsets of DR.SGX.

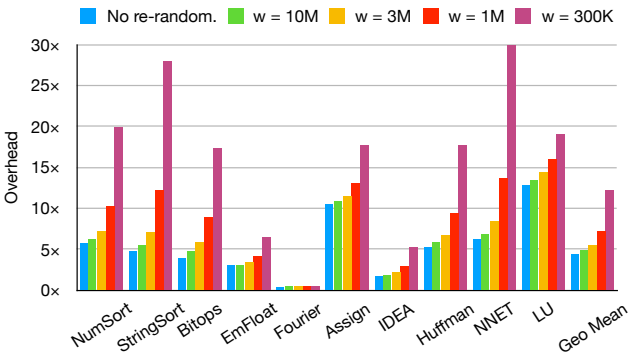

Figure 5: Overhead of each benchmark, with heap size h = 4 MB, without re-randomization and with various re-randomization windows w.

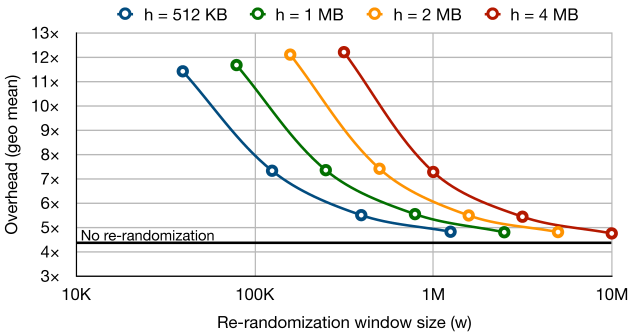

Figure 6: Geometric mean of the overheads, for various heap sizes h and various window sizes w. The black line represents the overhead without re-randomization.

for the code by factor n. We chose $n = 10$, thus the overhead is 10×. The size of the stack itself does not increase. For each invocation at run time only one of the function variants is used, i.e., the number of stack frames to be stored on the stack does not increase.

Runtime overhead of DR.SGX modifications. We evaluated DR.SGX with different subsets of its components active. To get a better understanding of the impact DR.SGX's individual components we ran all benchmarks multiple times activating one more components for every repetition. A breakdown of each component's overhead is shown in Figure 4.

We first tested our mechanism to move large stack allocations to the heap, i.e., replacing allocations on the stack larger than 63 bytes with calls to `malloc`. We measured a negligible overhead well below 1%, which is too small to be visible in Figure 4. Then, we tested the instrumentation of reads and writes (LLVM instruction `getelementptr`). In DR.SGX, instances of this instruction are followed by a call to our permutation function, unless the argument to the instruction is on the stack. In this test, the identity permutation function was used, which returns immediately. Therefore overhead reflects the impact of the instrumentation alone. We measured overheads between 0 and 102%, with a geometric mean of 39% (*GEP instr.* in Figure 4). Next, we added our stack randomization using function duplication. The geometric mean of the additional overhead is 46%, while the maximum is 135% (*Function 10×* in Figure 4). Finally, we tested our complete system (without periodic re-randomization). Overheads range between 0.39× and 12.77×, with a geometric mean of 4.36×. The benchmarks *Assign* and *LU* have the biggest overheads, 10.56× and 12.77× respectively, due to high miss rates in our permutation buffer (their miss rates are ~ 13× higher).

Runtime overhead of re-randomization. Next, we assessed the impact of various window sizes w and heap sizes h on the run time overhead and re-randomization window duration. We chose our heap size $h \in \{4\text{ MB}, 2\text{ MB}, 1\text{ MB}, 512\text{ KB}\}$ but other values are also possible. We measured the time required to perform a re-randomization by dividing the CPU cycles required by the processor's nominal speed, 3.4 GHz. The re-randomization requires 7.31 ms, 4.07 ms, 2.26 ms, and 1.26 ms respectively for $h = 4$ MB, $h = 2$ MB, $h = 1$ MB, $h = 512$ KB.

We first measured the run time overheads for $h = 4$ MB, $w \in \{10\text{ M}, 3\text{ M}, 1\text{ M}, 300\text{ K}\}$. In Figure 5, the left-most bars in each group represent the overhead without re-randomization (like Figure 4), with a geometric mean of 4.36×. Re-randomization every 10 million accesses ($w = 10$ M) increases the overhead slightly (geometric mean of 4.76×). Reducing the window to 3 M, 1 M, and 300 K brings the geometric mean of the overhead to 5.45×, 7.29×, and 12.21×.

We then measured the overhead for smaller heap sizes. We expected that halving both the heap size and the window size, i.e., re-randomizing a heap half as big twice as often, would yield similar performance results. Figure 6 shows the overhead depending on the window size for various heap sizes and confirms our intuition. Each line refers to a heap size twice as big as the line to its left. The black line at 4.26× is the overhead measured in the case without re-randomization and represents $\lim_{w\to\infty}$ of the overhead; in other words, increasing the values of w further would bring diminishing results.

Summary. The performance of our solution depends heavily on the user parameters. For example, the overhead is 4.8× for parameters h = 1MB and w = 2.5 M.

Developers and system administrators can adjust the parameters of DR.SGX based on the memory needs of their application

and the available computing resources. For example, if the deployment scenario requires 1 MB of heap memory and allows up to 8× overhead, the window size w can be set to 250 K for maximal re-randomization rate and security (see Figure 6). We consider this task of parameter tuning feasible for most developers. A typical developer may not be able to assess subtle sources of information leakage for correct source code annotation, but usually the developer knows the application's performance requirements and can set h and w accordingly.

Finally, we emphasize that in many SGX application scenarios, the overhead of the enclave (imposed by DR.SGX) is not directly the overhead of the entire application. For example, SGX-based applications that perform networking or database queries spend most of their time in the unprotected part of the application, and therefore the slowdown of the enclave represents only a minor part of the application's performance. Thus, in many cases, a high enclave overhead can still be acceptable for the overall performance.

6 SECURITY ANALYSIS

In this section we analyze the security of DR.SGX. We focus on the security properties of our novel heap data protection mechanism. Our stack data protection follows a known approach, evaluated in [19].

The goal of the adversary is to recover secret data from the victim enclave based on secret-dependent (heap) data access patterns to data. Recall that we consider a powerful adversary that gets a perfect trace of all cache and page events. Since all known attacks [11, 31, 51, 64] exhibit significant noise in the cache channel, this is an over-approximation of the capabilities of today's attackers and allows us to reason about the effectiveness of our solution against more powerful future adversaries.

In a data-driven side-channel attack, the adversary leaks information by monitoring secret-dependent access patterns. We model this as follows. The targeted victim enclave has secret data s of any length. The secret could be a cryptographic key, medical data, financial information or sensitive machine learning training sets. The enclave has a data structure d that consists of n elements $(e_1, ..., e_n)$ and is accessed based on s. The data structure could be a look-up table, S-box, index, or in-memory database. The size of each element e_i is the cache line size (smaller elements cannot be attacked, larger elements can be modeled as multiple elements). Based on the value of s, the enclave makes k accesses to different elements of d. Such access pattern determines the value of s. The enclave may also make predictable accesses to d (e.g., iterate through it during initialization).

6.1 Finding Attack Position in Trace

We start our analysis by explaining how the adversary can find the "attack position" in the side-channel trace, i.e., the position where (permuted) secret-dependent data accesses take place. The adversary can compile the victim enclave without DR.SGX protection and instrument those parts of the enclave where the secret-dependent accesses to d happen. The adversary can then run the instrumented enclave, monitor side-channels, and based on the instrumentation learn the position in the trace where the secret-dependent accesses are located. After that, the adversary can run the victim enclave that is protected with DR.SGX using the same inputs and again monitor side-channels. Assuming a deterministic enclave,[5] the adversary obtains a protected trace that includes additional randomization events to the trace (see Figure 7). Next, the adversary can filter out all randomization events. Since we use non-temporal (NT) writes that bypass the cache for randomization writes, the adversary finds each page event p_i that has no corresponding cache event c_i in the trace. For each such randomization write, the previous event in the trace is a read due to the randomization. The adversary removes all randomization events. The known attack position in the non-protected trace corresponds to the same position in the filtered protected trace.

6.2 Inferring Secret Enclave Data

Once the attack position is known, the adversary can attempt to infer secret data s from the permuted memory accesses in the attack trace. The adversary's success depends on the type of the victim enclave.

No predictable accesses. We first consider enclaves that make *no* predictable accesses to d (i.e., the enclave accesses d only based on a pattern that is derived from the secret data s). For such enclaves, DR.SGX provides strong protection due to its initial randomization that is illustrated in Figure 7. The enclave's data is copied from the known, original memory layout L_0 to a new randomized memory layout L_1 in blocks of cache line size using NT writes. For each block, the initial randomization process performs one read access to the original memory layout and NT writes to all memory pages. Because NT writes hide the accessed address at cache-line granularity, the adversary gains no knowledge of the new location in L_1. The same process is repeated for every memory block and in the end the location of each block in L_1 is equally likely for the adversary.

By observing the permuted side-channel trace, the adversary may infer execution characteristics such as frequencies of accesses to the same memory address (e.g., address a was accessed x times). However, because permuted addresses a can refer to any actually accessed addresses, such frequency analysis does not help the adversary to infer the secret data s, unless the enclave exhibits predictable access patterns which we discuss below.

Assuming no predictable access patterns, the best option for the adversary is a guessing attack. The adversary knows the permuted addresses of k secret-dependent accesses. For each access, every address, and thus every data structure element e_i, is equally likely. After observing k distinctive accesses to n elements, the number of possible alternatives will be given by an arrangement of k from n: $A_n^k = \frac{n!}{(n-k)!}$. For example, a data structure of $n = 50$ elements and any number of secret-dependent accesses resulting in 25 distinctive accesses to the data structure, the amount of arrangements is 1.96×10^{39}, which gives the chance of a random guess of approximately 2^{-131}. We conclude that DR.SGX provides strong protection for enclaves that have no predictable accesses to the data structure d.

[5] We consider a deterministic enclave, because that is the best case the for adversary for building the tracking tree. Thus, the following analysis based on this assumption represents the best case for the adversary regarding finding the attack position in the trace.

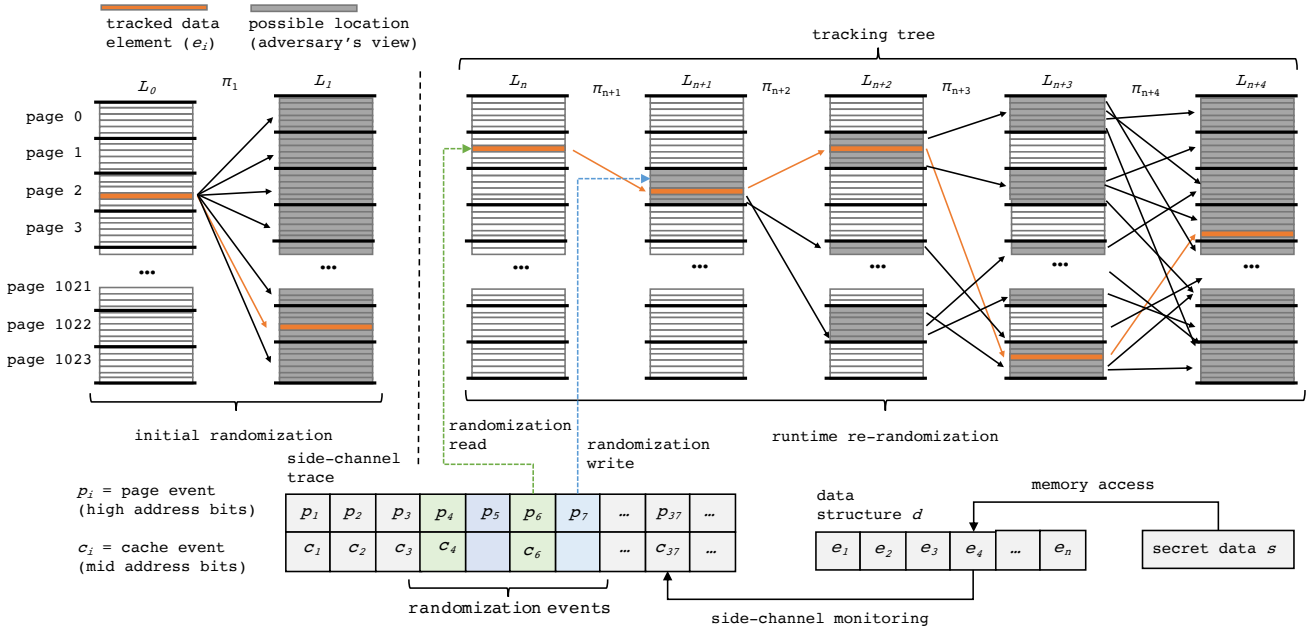

Figure 7: Location tracking. By identifying missing cache events in the trace, the adversary can learn the re-randomization writes (marked in blue) and preceding re-randomization reads (marked in green). The initial randomization hides destination addresses completely. The adversary can build a tracking tree, where the source address of each re-randomization is known with cache-line granularity and the destination address with page granularity. After few re-randomization rounds, the tracked memory location can reside in any memory location.

Predictable accesses. The second case that we consider is a victim enclave that exhibits predictable access patterns to d, e.g., the enclave may initialize d in an order that is known to the adversary. The enclave may also access elements of d a predictable number of times. Such predictable accesses in the trace will disclose the current permuted memory addresses for each accessed element e_i.

Figure 7 illustrates an example scenario, where the permuted address of element e_i is revealed to the adversary in memory layout L_n. The next re-randomization round moves the data of that element to a new location in layout L_{n+1}. Since the move operation is implemented using NT writes, the adversary learns the new page in L_{n+1}, but not the fine-grained location. The leakage of the target page allows the adversary to construct a *tracking tree* for element e_i.

The expansion of the tracking tree depends on the size of the used memory in the victim enclave. For example, if the victim enclave uses 2 MB memory (out of total 4 MB address space), each memory page contains on the average 32 blocks. On the next re-randomization round, each of these blocks are moved to new memory locations in layout L_{n+2}. Because the adversary does not know the exact location of element e_i in L_{n+1}, he cannot distinguish when the element is moved from the set of 32 move operations that use the same page as the source.[6] From the adversary's point of view, after two re-randomization rounds, the element can reside in 32 pages with high probability. After four re-randomization rounds, the adversary must track $32^3 = 32,768$ re-randomization moves. Although some of the moves may write to the same target pages, the tracking tree covers all 1,024 memory pages in L_{n+4} with high probability, and thus all memory locations are equally likely for the adversary. For enclaves with smaller heap size (512 KB), similar effect can be achieved after three rounds. The shortest re-randomization window we tested in Section 5 lasted 0.37 ms, in which case the required three (or four) re-randomization rounds would be performed after 1.1 ms (or 1.5 ms) of enclave execution. We conclude that enclaves with predictable accesses can leak information. If the secret-dependent access happens after the predictable access and before a sufficient number of re-randomization rounds, the secret may be leaked to the adversary. By touching additional memory pages on every re-randomization write, the window can be reduced to fewer rounds. Alternatively, re-randomization rounds can be performed more frequently. Both approaches increase runtime overhead.

[6]Our implementation randomizes 8 blocks at once which makes tracing even more difficult for the adversary.

7 RELATED WORK

Previous research has proposed various side-channel defenses. In this section we review them and compare existing defenses to DR.SGX.

ORAM and Oblivious Execution. Oblivious RAM (ORAM) [28–30, 60, 69, 76] refers to schemes that hide the memory access pattern of a trusted client (e.g., CPU or network client) to an untrusted and encrypted memory (e.g., DRAM or server) by introducing fake

accesses and shuffling the encrypted memory elements such that the observable access pattern is independent of the actual access pattern. Oblivious execution architectures [44, 45, 48] attempt to hide all observable effects of program execution, including both memory accesses (code and data) and timing information. Implementing ORAM for every enclave memory access is extremely expensive. Obfuscuro [1], a program obfuscation system, implements both ORAM and oblivious execution, with performance overheads of 83× on average and up to 220×. DR.SGX's performance overhead is at least one order of magnitude lower than Obfuscuro.

Sinha [67] proposes a compiler-based tool to protect code written in their custom language from paging-based side-channel attacks. In contrast, DR.SGX works with existing code in C/C++ and also mitigates cache-based side-channel attacks.

Raccoon [59] is a system that provides oblivious data access only for developer-annotated enclave data, thus reducing the overhead. Memory accesses are hidden by either using ORAM or by streaming over the entire data structure. In contrast, DR.SGX does not rely on developers to identify and annotate data that might leak.

ZeroTrace [62] is an oblivious data structure framework for SGX that runs on top of a software memory controller. ZeroTrace is designed to hide memory access to resources *outside* of an enclave, e.g., to the hard disk drive. Importantly, it is not designed to make *all* memory accesses of an enclave to its own main memory oblivious, like DR.SGX does. Furthermore, ZeroTrace requires the developer to use the memory controller interface for all access that should be protected. DR.SGX does not require similar developer assistance.

Ohrimenko et. al. propose data-oblivious machine learning algorithms [53] and a side-channel resilient MapReduce framework [52] for SGX. Fuhry et. al. propose a page-fault side-channel secure database [27]. Such defenses are tailored to specific enclaves and algorithms, while DR.SGX applies to arbitrary enclaves.

Transactional memory. Some of the known SGX side-channel attacks interrupt the victim enclave repeatedly [77]. A corresponding defense is to enable the victim enclave to detect interruption and take counteractive measures, such as stopping its execution. T-SGX [66] leverages the Intel Transactional Synchronization Extension (TSX) to detect asynchronous enclave exits, e.g., due to interrupts of page faults. Déjá Vu [17] monitors the execution time of an enclave to detect a slowdown caused by frequent interrupts. These defenses do not prevent attacks that work without interrupts [11, 31, 64]. DR.SGX is applicable to such attacks.

Cloak [33] uses TSX to preform atomic memory operations that hide sensitive memory accesses. Before sensitive memory is accessed, all cache lines are touched (primed) by the enclave, and thus the adversary learns nothing about the enclave's sensitive accesses. Cloak relies on the developer to annotate sensitive data structures that should be protected from side-channel attacks and requires TSX, which is not supported by all SGX processors. DR.SGX does not require similar developer assistance and works on all SGX processors.

Software diversity. Crane et al. [19] propose to apply dynamic software diversity, an effective countermeasure against code reuse attacks and reverse engineering, to defend against cache-based side-channel attacks. The approach is to create multiple copies of code and choose one of them at the time of execution. We apply this technique to protect stack data. However, the solution by Crane et al. is specifically targeting protection of cryptographic algorithms. In contrast, DR.SGX can protect non-cryptographic enclaves.

Randomization. Address Space Layout Randomization (ASLR) [57] is a common defensive technique against memory corruption attacks such as ROP [61]. ASLR hides the locations of memory regions (code and data) by randomizing their offsets at load time. More fine-grained solutions randomize code (but not data) at function [38], block [21, 75], or instruction [34, 56] level.

Such randomization techniques are insufficient as a side-channel defense for SGX. Offset-based ASLR is not effective since the privileged attacker is responsible for memory management and thus learns the "secret" randomized offsets. Code randomization, as implemented in SGX-Shield [65], is not complete [7] and does not prevent attacks that monitor data accesses [11, 64, 77].

New cache architectures. Cache-based side channels can be addressed by changes in the cache architecture. The two common approaches are (i) cache partitioning [23, 24, 55, 74], dividing the cache into partitions that are not shared between processes, and (ii) cache access obfuscation [22, 37, 40, 46, 74], where the goal is to obfuscate the obtainable side-channel information, either by introducing noise or by randomizing the address to cache line mapping. Such defenses require hardware changes and are limited to cache attacks. DR.SGX works on current processors and applies to additional side-channels (e.g., page faults).

8 DISCUSSION

Fine-grained leaks. Recent works [50, 78] have investigated the possibility of leaking information through a side-channel with a granularity smaller than a cache line. However, they are not applicable in our case.

CacheBleed [78] exploits cache *bank conflicts* to leak fine-grained information. This attack does not apply to SGX CPUs due to an updated cache design. We verified this experimentally.

MemJam [50] uses read-after-write false dependencies to introduce latency when a victim program reads data with a specific page offset. By measuring the run time of the victim program a high number of times while *jamming* different page offsets, the attacker can infer which offsets are read more often by the victim. This attack can leak information with a four byte granularity, but requires an extremely high number of runs (*50 million runs* for an attack against a simple and deterministic SGX enclave). However, with DR.SGX, the page offsets of data change between different runs, making the correlation of timing information for different runs exponentially more involved. Moreover, the accesses due to DR.SGX's own code generate a significant amount of noise, which complicates the matter further. Finally, the code of DR.SGX itself was designed to not be vulnerable to MemJam attacks, e.g., by randomizing the permutation buffer layout (see Section 4.5).

Leakage quantification. Quantification of cache-based information leakage has been studied in previous works. For example, CacheAudit [25] is a well-known static analysis framework that given an x86 binary and a cache configuration yields an upper

bound on the amount of information leakage via cache- and time-based side-channels. The information leakage is quantified based on the number of side-channel observations an attacker can obtain.

CacheAudit, and similar existing tools, are not applicable to our scenario for two main reasons. First, in the model of CacheAudit, randomly permuted observations contribute to the total number of observations, even though the attacker may not learn any useful information from such accesses. Second, CacheAudit does not consider information leakage through other channels, such as page faults, that can be correlated with cache observations. Therefore, CacheAudit cannot be used to quantify informations leakage of DR.SGX.

9 CONCLUSION

In this paper we have proposed semantic-agnostic data randomization as a new defensive approach against side-channel attacks on SGX. We have designed and implemented DR.SGX, which allows to instrument enclave code such that all data locations in enclave memory are permuted at cache-line granularity and re-randomized at runtime. Unlike previous defenses, our solution allows non-expert developers to harden their enclaves against various data-driven attack strategies with an adjustable security-performance trade-off.

ACKNOWLEDGMENTS

The authors would like to thank Urs Müller for his feedback in the initial discussions that led to this work.
This work has been supported by the German Research Foundation (DFG) as part of projects HWSec, P3 and S2 within the CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, by BMBF within the projects iBlockchain and CloudProtect, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

REFERENCES

[1] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. Obfuscuro: A Commodity Obfuscation Engine on Intel SGX. In *Network and Distributed System Security Symposium*.
[2] ARM Limited. 2009. ARM Security Technology – Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.
[3] Lejla Batina, Patrick Jauernig, Nele Mentens, A-R Sadeghi, and Emmanuel Stapf. 2019. In Hardware We Trust: Gains and Pains of Hardware-assisted Security. (2019).
[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven.
[5] Mihir Bellare, Phillip Rogaway, and Terence Spies. 2010. *The FFX Mode of Operation for Format-Preserving Encryption.* Technical Report.
[6] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM.
[7] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th USENIX Security Symposium*.
[8] Ferdinand Brasser, Lucas Davi, Abhijitt Dhavlle, Tommaso Frassetto, Sai Manoj Pudukotai Dinakarrao, Setareh Rafatirad, Ahmad-Reza Sadeghi, Avesta Sasan, Hossein Sayadi, Shaza Zeitouni, et al. 2018. Advances and throwbacks in hardware-assisted security: special session. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE Press, 15.
[9] Ferdinand Brasser, Tommaso Frassetto, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, and Christian Weinert. 2018. VoiceGuard: Secure and Private Speech Processing. In *Interspeech 2018*. International Speech Communication Association (ISCA), 1303–1307.
[10] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *26th Annual Network & Distributed System Security Symposium (NDSS)*.
[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies*.
[12] E. Brickell, G. Graunke, and J.-P. Seifert. 2006. Mitigating cache/timing attacks in AES and RSA software implementations. In *RSA Conference 2006, session DEV-203*.
[13] BYTE Magazine and Uwe F. Mayer. 1995-2011. BYTEmark benchmark (nbench), port to Linux. Original address http://www.tux.org/~mayer/linux/bmark.html, now archived at https://web.archive.org/web/20151215162836/http://www.tux.org/~mayer/linux/bmark.html.
[14] Luigi Catuogno, Alexandra Dmitrienko, Konrad Eriksson, Dirk Kuhlmann, Gianluca Ramunno, Ahmad-Reza Sadeghi, Steffen Schulz, Matthias Schunter, Marcel Winandy, and Jing Zhan. 2009. Trusted Virtual Domains – Design, Implementation and Lessons Learned. In *International Conference on Trusted Systems*.
[15] Swarup Chandra, Vishal Karande, Zhiqiang Lin, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. 2017. Securing Data Analytics on SGX with Randomization. In *European Symposium on Research in Computer Security*.
[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. arXiv:arXiv:1802.09085v3
[17] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *ACM Symposium on Information, Computer and Communications Security*.
[18] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained.* Technical Report. Cryptology ePrint Archive. Report 2016/086. https://eprint.iacr.org/2016/086.pdf.
[19] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Network and Distributed System Security Symposium*.
[20] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. 2019. FastKitten: Practical Smart Contracts on Bitcoin. In *28th USENIX Security Symposium*.
[21] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2013. Gadge Me If You Can - Secure and Efficient Ad-hoc Instruction-Level Randomization for x86 and ARM. In *ACM Symposium on Information, Computer and Communications Security*.
[22] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2020. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *29th USENIX Security Symposium*.
[23] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012).
[24] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable Caches: Low-complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization* (2012). https://doi.org/10.1145/2086696.2086714
[25] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)* 18, 1 (2015).
[26] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. JITGuard: Hardening Just-in-time Compilers with SGX. In *24th ACM Conference on Computer and Communications Security (CCS)*.
[27] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2017. HardIDX: Practical and Secure Index with SGX. In *Conference on Data and Applications Security and Privacy (DBSec)*.
[28] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Annual ACM Symposium on Theory of Computing*. ACM.
[29] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* (1996).
[30] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics.
[31] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*.
[32] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity18/presentation/gras
[33] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium*.

[34] Jason D. Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd My Gadgets Go?. In *IEEE Symposium on Security and Privacy*.

[35] Intel. 2015. Intel Software Guard Extensions. Tutorial slides. https://software.intel.com/sites/default/files/332680-002.pdf. Reference Number: 332680-002, revision 1.1.

[36] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[37] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. 2008. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems* (2008).

[38] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Annual Computer Security Applications Conference*.

[39] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[40] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. 2009. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *IEEE International Symposium on High Performance Computer Architecture*. IEEE.

[41] Kubilay Ahmet Küçük, Andrew Paverd, Andrew Martin, N. Asokan, Andrew Simpson, and Robin Ankele. 2016. Exploring the Use of Intel SGX for Secure Many-Party Applications.

[42] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*.

[43] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[44] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGARCH Computer Architecture News* 43, 1 (2015).

[45] Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory trace oblivious program execution. In *IEEE Computer Security Foundations Symposium*.

[46] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*.

[47] LLVM Foundation. 2019. The LLVM Compiler Infrastructure. https://llvm.org.

[48] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. Phantom: Practical oblivious computation in a secure processor. In *ACM SIGSAC Conference on Computer and Communications Security*.

[49] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*.

[50] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *Topics in Cryptology – CT-RSA 2018*, Nigel P. Smart (Ed.). Springer International Publishing.

[51] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. *CacheZoom: How SGX Amplifies The Power of Cache Attacks*. Technical Report. arXiv:1703.06986 [cs.CR]. https://arxiv.org/abs/1703.06986.

[52] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. 2015. Observing and preventing leakage in MapReduce. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[53] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aasthaa Meht, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium*.

[54] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology*.

[55] D. Page. 2005. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. In *IACR Eprint archive*.

[56] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *IEEE Symposium on Security and Privacy*.

[57] PaX Team. [n.d.]. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[58] Bernardo Portela, Manuel Barbosa, Guillaume Scerri, Bogdan Warinschi, Raad Bahmani, Ferdinand Brasser, and Ahmad-Reza Sadeghi. 2017. Secure Multiparty Computation from SGX. In *Financial Cryptography and Data Security*.

[59] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *USENIX Security Symposium*. http://dl.acm.org/citation.cfm?id=2831143.2831171

[60] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security Symposium*.

[61] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security* 15, 1 (2012).

[62] Sajin Sasy, Sergey Gorbunov, and Christopher Fletcher. 2017. ZeroTrace: Oblivious Memory Primitives from Intel SGX. *IACR Cryptology ' Archive* Report 2017/549 (2017).

[63] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX.

[64] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[65] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Network and Distributed System Security Symposium*.

[66] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium*.

[67] Rohit Sinha, Sriram Rajamani, and Sanjit A. Seshia. 2017. A compiler and verifier for page access oblivious computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ACM Press. https://doi.org/10.1145/3106237.3106248

[68] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *CoRR* abs/1806.07480 (2018). arXiv:1806.07480 http://arxiv.org/abs/1806.07480

[69] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM SIGSAC Conference on Computer and Communications Security*.

[70] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference*.

[71] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium*.

[72] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium*.

[73] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *27th USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity18/presentation/van-schaik

[74] Zhenghong Wang and Ruby B. Lee. 2008. A Novel Cache Architecture with Enhanced Performance and Security. In *Annual IEEE/ACM International Symposium on Microarchitecture*.

[75] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *ACM SIGSAC Conference on Computer and Communications Security*.

[76] Peter Williams and Radu Sion. 2012. Round-optimal access privacy on outsourced storage. In *ACM Conference on Computer and Communications Security*.

[77] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy*.

[78] Y. Yarom, D. Genkin, and N. Heninger. 2016. *CacheBleed: A timing attack on OpenSSL constant time RSA*. Technical Report. Cryptology ePrint Archive. Report 2016/224. https://eprint.iacr.org/2016/224.pdf.

FASTKITTEN: Practical Smart Contracts on Bitcoin

Poulami Das* Lisa Eckey* Tommaso Frassetto§ David Gens§
Kristina Hostáková* Patrick Jauernig§ Sebastian Faust* Ahmad-Reza Sadeghi§
Technische Universität Darmstadt, Germany
** first.last@cs.tu-darmstadt.de*
§ first.last@trust.tu-darmstadt.de

Abstract

Smart contracts are envisioned to be one of the killer applications of decentralized cryptocurrencies. They enable self-enforcing payments between users depending on complex program logic. Unfortunately, Bitcoin – the largest and by far most widely used cryptocurrency – does not offer support for complex smart contracts. Moreover, simple contracts that can be executed on Bitcoin are often cumbersome to design and very costly to execute. In this work we present FASTKITTEN, a practical framework for executing arbitrarily complex smart contracts at low costs over decentralized cryptocurrencies which are designed to only support simple transactions. To this end, FASTKITTEN leverages the power of trusted computing environments (TEEs), in which contracts are run off-chain to enable efficient contract execution at low cost. We formally prove that FASTKITTEN satisfies strong security properties when all but one party are malicious. Finally, we report on a prototype implementation which supports arbitrary contracts through a scripting engine, and evaluate performance through benchmarking a provably fair online poker game. Our implementation illustrates that FASTKITTEN is practical for complex multi-round applications with a very small latency. Combining these features, FASTKITTEN is the *first* truly practical framework for complex smart contract execution over Bitcoin.

1 Introduction

Starting with their invention in 2008, decentralized cryptocurrencies such as Bitcoin [51] currently receive broad attention both from academia and industry. Since the rise of Bitcoin, countless new cryptocurrencies have been launched to address some of the shortcomings of Nakamoto's original proposal. Examples include Zerocash [47] which improves on Bitcoin's limited anonymity, and Ethereum [16] which offers complex smart contract support. Despite these developments, Bitcoin still remains by far the most popular and intensively studied cryptocurrency, with its current market capitalization of $109 billion which accounts for more than 50% of the total cryptocurrency market size [2].

A particular important shortcoming of Bitcoin is its limited support for so-called smart contracts. Smart contracts are (partially) self-enforcing protocols that allow emitting transactions based on complex program logic. Smart contracts enable countless novel applications in, e.g., the financial industry or for the Internet of Things, and are often quoted as a glimpse into our future [9]. The most prominent cryptocurrency that currently allows to run complex smart contracts is Ethereum [16], which has been designed to support Turing complete smart contracts. While Ethereum is continuously gaining popularity, integrating contracts directly into a cryptocurrency has several downsides as frequently mentioned by the advocates of Bitcoin. First, designing large-scale secure distributed systems is highly complex, and increasing complexity even further by adding support for complex smart contracts also increases the potential for introducing bugs. Second, in Ethereum, smart contracts are directly integrated into the consensus mechanics of the cryptocurrency, which requires in particular that all nodes of the decentralized system execute all contracts. This makes execution of contracts very costly and limits the number and complexity of applications that can eventually be run over such a system. Finally, many applications for smart contracts require confidentiality, which is currently not supported by Ethereum.

There has been significant research effort in addressing these challenges individually. Some works aim to extend the functionality of Bitcoin by showing how to build contracts over Bitcoin by using multiparty computation (MPC) [37, 38, 40], others focus on achieving privacy-preserving contracts (e.g., Hawk [35], Ekiden [19]) by combining existing cryptocurrencies with trusted execution environments (TEEs). However, as we elaborate in Section 2, all of these solutions suffer from various deficiencies: they cannot be integrated into existing cryptocurrencies such as Bitcoin, are highly inefficient (e.g., they use heavy cryptographic techniques such as non-interactive zero-knowledge proofs or general MPC), do not support money mechanics, or have significant financial costs due to complex transactions and high collateral (money blocked by the parties in MPC-based solutions).

In this work, we propose FASTKITTEN, a novel system that leverages trusted execution environments (TEEs) utilizing well-established cryptocurrencies, such as Bitcoin, to offer full support for arbitrary complex smart contracts. We emphasize that FASTKITTEN does not only address the challenges discussed above, but is also highly efficient. It can be easily integrated into existing cryptocurrencies and hence is ready to use today. FASTKITTEN achieves these goals by using a TEE to isolate the contract execution inside an enclave, shielding it from potentially malicious users. The main challenges of this solution, such as for instance how to load and validate blockchain data inside the enclave or how to prevent denial of service attacks, are discussed in Section 3.1. Moving the contract execution into the secure enclave guarantees correct and private evaluation of the smart contract even if it is not running on the blockchain and verified by the decentralized network. This approach circumvents the efficiency shortcoming of cryptocurrencies like Ethereum, where contracts have to be executed in parallel by thousands of users. Most related to our work is the recently introduced Ekiden system [19], which uses a TEE to support execution of multiparty computations but does support contracts that handle coins. While Ekiden is efficient for single round contracts, it is not designed for complex reactive multi-round contracts, and their off-chain execution. The latter is one of the main goals of FASTKITTEN.

We summarize our main goals and contributions below.

- **Smart Contracts for Bitcoin:** We support arbitrary multi-round smart contracts executed amongst any finite number of participants, where our system can be run on top of any cryptocurrency with only limited scripting functionality. We emphasize that Bitcoin is only one example over which our system can be deployed today; even cryptocurrencies that are simpler than Bitcoin can be used for FASTKITTEN.
- **Efficient Off-Chain Execution:** Our protocol is designed to keep the vast majority of program execution off-chain in the standard case if all parties follow the protocol. Since our system incentivizes honest behavior for most practical use cases, FASTKITTEN can thus run in real-time at low costs.
- **Formal Security Analysis:** We formally analyze the security of FASTKITTEN in a strong adversarial model. We prove that either the contract is executed correctly, or all honest parties get their money back that they have initially invested into the contract, while a malicious party loses its coins. Additionally, the service provider who runs the TEE is provably guaranteed to not lose money if he behaves honestly.
- **Implementation and benchmarking:** We provide an in-depth analysis of FASTKITTEN's performance and costs and evaluate our framework implementation with respect to several system parameters by offering benchmarks on real-world use cases. Concretely, we show that online poker can run with an overall match latency of 45ms and costs per player are in order of magnitude of one USD, which demonstrates FASTKITTEN's practicality.

We emphasize that FASTKITTEN requires only a single TEE which can be owned either by one of the participants or by an external service provider which we call the *operator*. In addition, smart contracts running in the FASTKITTEN execution framework support private state and secure inputs, and thus, offer even more powerful contracts than Ethereum. Finally, we stress that FASTKITTEN can support contracts that may span over multiple different cryptocurrencies where each participant may use her favorite currency for the money handled by the contract.

2 Related Work

Support for execution of arbitrary complex smart contracts over decentralized cryptocurrencies was first proposed and implemented by the Ethereum cryptocurrency. As pointed out in Section 1, running smart contracts over decentralized cryptocurrencies results in significant overheads due to the replicated execution of the contract. While there are currently huge research efforts aiming at reducing these overheads (for instance, via second layer solutions such as state channels [24, 49], Arbitrum [34] or Plasma [55], outsourcing of computation [58], or permissioned blockchains [46]), these solutions work only over cryptocurrencies with support complex smart contracts, e.g. over Ethereum. Another line of work, which includes Hawk [36] and the "Ring of Gyges" [33], is addressing the shortcoming that Ethereum smart contracts cannot keep private state. However, also these solutions are based on complex smart contracts and hence cannot be integrated into popular legacy cryptocurrencies such as Bitcoin, which is the main goal of FASTKITTEN.

In this section we will focus on related work, which considers smart contract execution on Bitcoin. We separately discuss multiparty computation based smart contracts and solutions using a TEE. We provide a more detailed discussion on how the above-mentioned Ethereum based solutions compare to FASTKITTEN in Appendix A. Additionally, in Section 8 we discuss some exemplary contract use cases and compare their execution inside FASTKITTEN with the execution over Ethereum.

Multiparty computation for smart contracts An interesting direction to realize complex contracts over Bitcoin is to use so-called multiparty computation with penalties [38–40]. Similar to FASTKITTEN these works allow secure m-round contract execution but they rely on the claim-or-refund functionality [39]. Such a functionality can be instantiated over Bitcoin and hence these works illustrate feasibility of generic contracts over Bitcoin. Unfortunately, solutions supporting generic contracts require complex (and expensive) Bitcoin transactions and high collateral locked by the parties which makes them impractical for most use-cases. Concretely, in

Approach	Minimal # TX	Collateral	Generic Contracts	Privacy
Ethereum contracts	$\mathcal{O}(m)$	$\mathcal{O}(n)$	✓	×
MPC [38–40]	$\mathcal{O}(1)$	$\mathcal{O}(n^2m)$	✓	✓
Ekiden [19]	$\mathcal{O}(m)$	no support for money		✓
FastKitten	$\mathcal{O}(1)$	$\mathcal{O}(n)$	✓	✓

Table 1: Selected solutions for contract execution over Bitcoin and their comparison to Ethereum smart contracts. Above, n denotes the number of parties and m is the number of reactive execution rounds.

all generic n-party contract solutions we are aware of, each party needs to lock $\mathcal{O}(nm)$ coins, which overall results in $\mathcal{O}(n^2m)$ of locked collateral. In contrast, the total collateral in FastKitten is $\mathcal{O}(n)$, see column "Collateral" in Table 1. It has been shown that for specific applications, concretely, a multi-party lottery, significant improvements in the required collateral are possible when using MPC-based solutions [48]. This however comes at the cost of an inefficient setup phase, communication complexity of order $\mathcal{O}(2^n)$, and $\mathcal{O}(\log n)$ on-chain transactions for the execution phase. Let us stress that the approach used in [48] cannot be applied to generic contracts.

Overall, while MPC-based contracts are an interesting direction for further research, we emphasize that these systems are currently far from providing a truly practical general-purpose platform for contract execution over Bitcoin—which is the main goal of FastKitten.

TEEs for blockchains There has recently been a large body of work on using TEEs to improve certain features of blockchains [10, 43, 59, 63, 64]. A prominent example is Teechain [43], which enables off-chain payment channel systems over Bitcoin. Most of these prior works do not use the TEE for smart contract execution. Some notable exceptions include Hawk [36] and the "Ring of Gyges" [33], who propose privacy preserving off-chain contracts execution, but, as already mentioned, do not work over Bitcoin.

Probably most related to our work is Ekiden [19], which proposes a system for private off-chain smart contract execution using TEEs. While Ekiden focuses on solutions over Ethereum, it does not require a powerful scripting language of the underlying blockchain technology – just like FastKitten. Despite the conceptual similarities of Ekiden and FastKitten, the goals of these systems are orthogonal. Ekiden aims at moving heavy smart contract execution off the chain in order to reduce the cost of executing complex contract functions. In contrast, FastKitten focuses on efficient off-chain execution of multi-round contracts between a set of parties. Importantly, we require our system to natively handle coins of the underlying blockchain. A joint goal of both systems is to provide state privacy of the contracts.

Ekiden considers clients (contract parties) and computing nodes which have a similar task as FastKitten's TEE operator since they also execute contracts inside a TEE. In contrast to FastKitten, Ekiden sends the encryption of the resulting contract state to the blockchain after every function call. If a client requests another function call, a selected computing node takes the state from the blockchain, decrypts it inside its enclave and performs the contract execution. This implies that reactive multi-round contracts are very costly even in the standard case when all participating parties are honest (c.f. column "Minimal # TX" in Table 1).

Ekiden relies on multiple TEEs and guarantees service availability as long as at least one TEE is controlled by an honest computing node. We note in Section 9.2 that fault tolerance can be integrated into FastKitten in a straightforward way. Additionally, Ekiden aims to achieve forward secrecy even if a small fraction of TEEs gets corrupted via, e.g., a side-channel attack. Their strategy is to secret-share a long-term secret key between the TEEs and use it to generate a short-term secret key every "epoch". Hence, an attacker learning the short-term key can only decrypt state from the current epoch. While side-channel attacks are out of scope of this work, note that FastKitten can achieve forward secrecy of states in case of side-channel attacks using the same mechanism as Ekiden.

An important part of the FastKitten construction is the fair distribution of coins through the enclave. Ekiden does neither model nor discuss the handling of coins. It is not straightforward to add this feature to their model since the contract state is encrypted and hence the money cannot be unlocked automatically on-chain.

3 Design

FastKitten allows a set of n users $P_1, \ldots, P_n$ to execute an arbitrary complex smart contract over a decentralized cryptocurrency that only supports very simple scripts. Concretely, FastKitten considers cryptocurrencies that, in addition to supporting simple transactions between users, offer so-called *time-locked transactions*. A transaction is time-locked if it is only processed and integrated into the blockchain after a certain amount of time has passed. Moreover, FastKitten requires that transactions contain space for storing arbitrary raw data. We emphasize that these are very mild requirements on the underlying cryptocurrency that, for instance, are satisfied by the most prominent cryptocurrency Bitcoin.[1] FastKitten leverages these properties together with the power of trusted execution environments to provide an efficient general-purpose smart contract execution platform.

As discussed in the introduction, a contract is a program that handles coins according to some—possibly complex—program logic. In this work, we consider n-party contracts, which are run among a group of parties $P_1, \ldots, P_n$ and have the following structure. During the initialization phase, the contract receives coins from the parties and some initial in-

[1] Bitcoin transactions can store up to 97 KB of data [44]; multiple transactions can be used for bigger payloads.

puts. Next, it runs for m reactive rounds, where in each round the contract can receive additional inputs from the parties P_i, and produces an output. Finally, after the m-th round is completed the contract pays out the coins to the parties according to its final state and terminates.

A key feature of FASTKITTEN is very low execution cost and high performance compared to contract execution over cryptocurrencies such as Ethereum. This is achieved by not executing contracts *by all parties maintaining the cryptocurrency* but instead running the contract within a TEE which could, e.g., be owned and operated by a single service provider which we call the *operator* Q. In the standard case when all parties are honest, FASTKITTEN runs the entire contract off-chain within the enclave and only needs to touch the blockchain during contract initialization and finalization. More concretely, during initialization, the parties transfer their coins to the enclave by time-locking coins with *deposit transactions*, while at the end of finalization the enclave produces transactions that transfer coins back to the users according to the results of the contract execution. These transactions are called *output transactions* and can be published by the users of the system to receive their coins.

3.1 Design Challenges of FASTKITTEN

Leveraging TEEs for building a general-purpose contract execution platform requires us to resolve the following main challenges.

Protection against malicious operator. The operator runs the TEE and hence controls its interaction with the environment (e.g., with other parties or the blockchain). Thus, the operator can abort the execution of the TEE, delay and change inputs, or drop any ingoing or outgoing message. To protect honest users from such an operator, the enclave program running inside the TEE must identify such malicious behavior and punish the operator. In particular, we require that even if the TEE execution is aborted, all parties must be able to get their coins refunded eventually. To achieve this, we let the operator create a so-called *penalty transaction*: the penalty transaction time-locks coins of the operator, which in case of misbehavior can be used to refund the users and punish the operator.

Note that designing such a scheme for punishment is highly non-trivial. Consider a situation where party P_i was supposed to send a message x to the contract. From the point of view of the enclave that runs the contract, it is not clear whether the operator was behaving maliciously and did not forward a message to the enclave, or, e.g., party P_i did not send the required message to the operator. To resolve this conflict, we leverage a challenge-response mechanism carried out via the blockchain. We emphasize that this challenge-response mechanism is only required when parties are malicious, and typically will not be executed often due to the high financial costs for an adversary.

Verification of blockchain evidence. To ensure that a malicious operator cannot make up false blockchain evidence, we need to design a secure blockchain validation algorithm which can efficiently be executed inside a TEE. We achieve this by simplifying the verification process typically carried out by full blockchain nodes by using a *checkpoint block* to serve as the initial starting point for verification. This drastically reduces blockchain verification time in comparison to verification starting from the genesis block. To further speed up the transaction verification, we only validate correctness of block headers. Finally, when the TEE needs to verify whether a certain transaction was integrated into a block, we set a minimum number of blocks that must confirm a transaction as part of the security parameter within our protocol. This guarantees that faking a valid-looking chain is computationally infeasible for a malicious operator. Finally, it is computationally infeasible for a malicious operator to load a fake (but valid-looking) chain into the enclave before the penalty transaction is published on the blockchain.

Minimizing blockchain interaction. Since blockchain interactions are expensive, FASTKITTEN only requires interaction with the blockchain in the initialization and finalization phases if all parties follow the protocol. As already discussed above, however, in case of malicious behavior FASTKITTEN may require additional interaction with the blockchain for conflict resolution. This is required to allow the TEE to attribute malicious behavior either to the operator or to some other participant P_i that provides input to the contract. We achieve this through a novel challenge-response protocol, where the TEE will ask the operator to challenge P_i via the blockchain. The operator can then either deliver a proof that he challenged P_i via the blockchain but did not receive a response, in which case P_i will get punished; or the operator receives P_i's input and can continue with the protocol.

Of course, this challenge-response protocol adds to the worst-case execution time of our system, and additionally will result in fees for blockchain interaction. To address the latter, our protocol ensures that both parties involved in the challenge-response mechanism have to split the fees resulting from blockchain interaction equally.[2] This incentivizes honest behavior if parties aim to maximize their personal profits.

Preventing denial of service attacks. Complex smart contracts may take a very long time to complete, and in the worst case not terminate. Hence, a malicious party may carry out a denial-of-service attack against the contract execution platform, where the platform is asked to execute a contract that never halts. It is well known that determining whether a program terminates is undecidable. Hence, general-purpose contract platforms, such as Ethereum, mitigate this risk by letting users pay via fees for every step of the contract execution. This effectively limits the amount of computation that

[2] In the cryptocurrency community, this is often referred to as griefing factor 1 : 1, meaning that for every coin spent by the honest users on fees the adversary is required to also spend one coin.

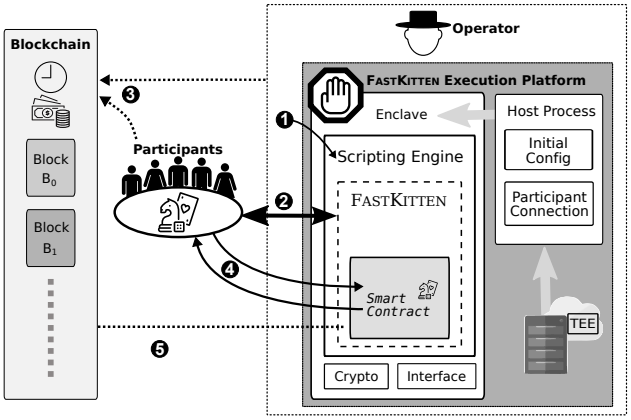

Figure 1: Architecture of the **FastKitten** Smart Contract Execution Platform. Dashed arrows indicate interaction with the blockchain and non-dashed arrows depict communication between parties.

can be carried out by the contract. Since FastKitten allows multiple parties to provide input to the contract in the same round, it might be impossible to decide which party (parties) caused the denial of service and should pay the fee. To this end, FastKitten protects against such denial-of-service attacks using a time-out mechanism. As all users of the system (including the operator) have to agree on the contract to be executed, we assume that this agreement includes a limit on the maximum amount of execution steps that can be performed inside the enclave per one execution round. See Section 6.5 for more details.

3.2 Architecture and Protocol

To enable secure off-chain contract execution, our architecture builds on existing TEEs, which are widely available through commercial off-the-shelf hardware. In particular, our architecture can be implemented using Intel's Software Guard Extensions (SGX) [4, 29, 45] which is a prominent TEE instantiation built into most recent Intel processors. SGX incorporates a set of new instructions to create, control and communicate with enclaves. While enclaves are part of a legacy host process, SGX enforces strict isolation of computation and memory between enclave and host process on the hardware level. Another prominent instantiation of the TEE concept is ARM TrustZone [6], which provides similar functionality for mobile devices. We note that only the operator Q is required to own TEE-enabled hardware.

As depicted in Figure 1, our FastKitten Execution Facility is run by the operator Q and consists of a host process and an enclave. The untrusted host process takes care of setting up the enclave with an *initial config*, handles the *participant connections*, and blockchain communication over the network. While this means that Q has complete control over these parts, the influence of a malicious operator on a running enclave is limited: he can interrupt enclave execution, but not tamper with it. Further, the enclave will sign and hash all code and data as part of its *attestation* towards parties, so they can verify correctness of the setup before placing deposits. To support arbitrary contract functionality, FastKitten includes a *scripting engine* inside the enclave and several helper libraries, such as the *Crypto library* to generate and verify transactions, and an *Interface library* to pass data between host process and enclave. The individual contracts are loaded into the FastKitten enclave during the initialization of our protocol by the underlying host process and participants can verify that contracts are loaded correctly. Our protocol then proceeds in three phases, which we call *setup phase*, *round computation*, and *finalization phase*. Figure 1 depicts the architecture of the FastKitten framework.

During the setup phase (Steps ❶–❸) the contract is loaded into the enclave. Using the TEE's attestation functionality, all parties $P_1, \ldots, P_n$ can verify that this step was completed correctly. Then the operator and all parties block their coins for the contract execution. If any party aborts in this phase, the money is refunded to all parties that deposited money and the protocol stops. Otherwise, all parties receive a time-locked penalty transaction, needed in case Q aborts the protocol.

Afterwards, the round computation phase (Step ❹) starts, in which Q sends the previous round's output to all parties. If a party P_i receives such an output, which is correctly signed by the enclave, it signs and sends the input for the following round to Q. If all parties behave honestly, Q will forward the received round inputs to the enclave, which computes the outputs for the next round. In case that the enclave does not receive an input from party P_i the enclave needs to determine whether P_i failed to send its input or if Q behaved maliciously (e.g., by dropping the message). Therefore, the enclave will punish Q unless it can prove, that it sent the last round output to P_i but did not receive a response. This proof is generated via the blockchain: Q publicly challenges P_i to respond with the input for the next round by posting the output of the previous round to the blockchain. As soon as this challenge transaction is confirmed, P_i needs to respond publicly by spending the coins of the challenge transaction and include its input for the next round. If P_i responds, Q can extract P_i's input and continue with the protocol execution. If P_i did not respond, Q forwards the respective blocks as a transcript to the enclave, to prove that P_i misbehaved.[3] So, while a malicious party (or the operator) can force this on-chain challenge-response procedure without direct punishment, posting these transactions will also act against its own financial interests by extending the time lock of its own coins and leading to transaction fees. Nevertheless, such malicious behavior cannot prevent the fair termination of our protocol.

The last phase of the protocol is the payout phase (Step ❺). In

[3] Alternatively, we could allow the operator to spend the challenge transaction after a timeout has passed. While this would result in easier verification for the TEE, the operator would need to publish an additional transaction, increasing both fees and the overall time for the challenge-response phase.

this phase the enclave returns the output transaction generated by the *Crypto library*. This transaction distributes the coins according to the terminated contract. In case of a protocol abort, the coins initially put by the users will be refunded to all honest parties. If any party was caught cheating, this party will not receive back its coins. This means the money will stay in control of the enclave and will never be spent.

4 Adversary Model

The FASTKITTEN protocol is executed n parties $P_1, \ldots, P_n$ and an operator Q (who owns the TEE) with the goal of executing a smart contract C. FASTKITTEN's design depends on a TEE to ensure its confidentiality and integrity. Our design is TEE-agnostic, even if our implementation is based on Intel SGX. Recent research showed that the security and privacy guarantees of SGX can be affected by memory-corruption vulnerabilities [11], architectural [13] and micro-architectural side-channel attacks [60]. For the operator, we assume that Q has full control over the machine and consequently can execute arbitrary code with supervisor privileges. While memory corruption vulnerabilities can exist in the enclave code, a malicious operator must exploit such vulnerabilities through the standard interface between the host process and the enclave. For the enclave code, we assume a common code-reuse defense such as control-flow integrity (CFI) [3, 15], or fine-grained code randomization [23, 42] to be in place and active. Architectural side-channel attacks, e.g., based on caches, can expose access patterns [13] from SGX enclaves (and therefore our FASTKITTEN prototype). However, this prompted the community to develop a number of software mitigations [12, 18, 27, 56, 57] and new hardware-based solutions [22, 28, 52]. Microarchitectural side-channel attacks like Foreshadow [60] can extract plaintext data and effectively undermine the attestation process FASTKITTEN relies on, leaking secrets and enabling the enclave to run a different application than agreed on by the parties; however, the vulnerability enabling Foreshadow was already patched by Intel [32]. Since existing defenses already target SGX vulnerabilities and since FASTKITTEN's design is TEE agnostic (i.e., it can also be implemented using ARM TrustZone or next-generation TEEs), we consider mitigating side-channel leakage as an orthogonal problem and out of scope for this paper.

For our protocol we consider a *byzantine adversary* [41], which means that corrupted parties can behave arbitrarily. In particular, this includes aborting the execution, dropping messages, and changing their inputs and outputs even if it means that they will lose money. FASTKITTEN is secure even if n parties are corrupt (including the two cases where only the operator is honest, and only one party is honest but the operator is corrupt). We show that no honest party will lose coins, a corrupt party will be penalized and that no adversary can tamper with the result of the contract execution. While we prove security in this very strong adversarial model, we additionally observe that incentive-driven parties (i.e., parties that aim at maximizing their financial profits) will behave honestly, which significantly boosts efficiency of our scheme. We stress that security of FASTKITTEN relies on the security of the underlying blockchain. We require that the underlying blockchain systems satisfies three security properties: *liveness*, *consistency* and *immutability* [26]. *Liveness* means that valid transactions are guaranteed to be included within the next δ blocks. *Consistency* guarantees that eventually all users have the same view on the current state of the blockchain (i.e., the transactions processed and their order). In addition, blockchains also are *immutable*, which means that once transactions end up in the blockchain they cannot be reverted. Most blockchain based cryptocurrencies guarantee consistency and immutability only after some time has passed, where time is measured by so-called *confirmations*. A block b_i is confirmed k-times if there exists a valid chain extending b_i with k further blocks. Once block b_i has been sufficiently often confirmed, we can assume that the transactions in b_i cannot be reverted and all honest parties agree on an order of the chain $(b_0, b_1, b_2, \ldots, b_i)$. For most practical purposes k can be a small constant, i.e., in Bitcoin it is generally believed that for $k = 6$ a block can be assumed final.[4]

5 The FASTKITTEN Protocol

In this section we give a more detailed description of our protocol, which includes the specification of the protocol run by Q and honest parties $P_1, \ldots, P_n$, all transactions and a description of the enclave program FASTKITTEN. The interaction between Q, P_i and the blockchain is depicted in Figure 2. We first describe the interactions with the blockchain and TEE.

5.1 Modeling the Blockchain

We will introduce some basic concepts of cryptocurrencies that are relevant for our work before we describe our high-level design. Cryptocurrencies are built using blockchains—a distributed data structure that is maintained by special parties called *miners*. The blockchain is comprised as a chain of blocks $(b_0, b_1, b_2, \ldots)$ that store the transactions of the system. The miners create new blocks by verifying new transactions and comprising them into new blocks that extend the tail of the chain. New blocks are created within some period of time t, where, for instance, in Bitcoin a new valid block is created every 10 minutes on average.

In cryptocurrencies users are identified by addresses, where an address is represented by a public key. To send coins from one address to another, most cryptocurrencies rely on transactions. If a user A with address pk_A wants to send x coins to user B with address pk_B, she creates a transaction $\mathtt{tx}$ which states that x coins from address pk_A are transferred to pk_B. Such a

[4] We notice that in blockchain-based cryptocurrencies there is no guaranteed finality, and even for very large values of k blocks can be reverted in principle. We emphasize however that even for small values of k reverting blocks becomes impossible in practice very quickly.

transaction tx is represented by the following tuple:

$$\mathsf{tx} := (\mathsf{tx.Input}, \mathsf{tx.Output}, \mathsf{tx.Time}, \mathsf{tx.Data}),$$

where tx.Input refers to a previously unspent transaction, tx.Output denotes the address to which tx.Value are going to be transferred to. Note that a transaction tx is unspent if it is not referred to by any other transaction in its Input field. Further, $\mathsf{tx.Time} \in \mathbb{N}$, which denotes the block counter after which this transaction will be included by miners, i.e., tx can be integrated into blocks $b_i, b_{i+1}, ...$, where $i = \mathsf{tx.Time}$. Finally, $\mathsf{tx.Data} \in \{0,1\}^*$ is a data field that can store arbitrary raw data. Similar to [5], we will often represent transactions by tables as shown exemplary in the table below, where the first row of the table gives the name of the transaction.

Transaction tx	
tx.Input:	Coins from unspent input transaction
tx.Output:	Coins to receiver address
tx.Time:	Some timelock (optional)
tx.Data:	Some data (optional)

Notice that a transaction tx only becomes valid if it is signed with the corresponding secret key of the output address from tx.Input. We emphasize that the properties described above are very mild and are for instance achieved by the most prominent cryptocurrency Bitcoin.

In order to model interaction with the cryptocurrency, we use a simplified blockchain functionality BC, which maintains a continuously growing chain of blocks. Internally it stores a block counter c which starts initially with 0 and is increased on average every t minutes. Every time the counter is increased, a new block will be created and all parties are notified. To address the uncertainty of the block creation duration we give the adversary control over the exact time when the counter is increased but it must not deviate more than $\Delta \in [t-1]$ seconds from t. Whenever any party publishes a valid transaction, it is guaranteed to be included in any of the next δ blocks.

Parties can interact with the blockchain functionality BC using the following commands.

- BC.post(tx): If the transaction tx is valid (i.e., all inputs refer to unspent transactions assigned to creator of tx and the sum of all output coins is not larger than the sum of all input coins) then tx is stored in any of the blocks $\{b_{c+1}, \ldots, b_{c+\delta}\}$.
- BC.getAll(i): If $i < c$, this function returns the latest block count $c-1$ and a list of blocks that extend b_i: $\mathbf{b} = (b_{i+1}, \ldots, b_c)$
- BC.getLast(): The function getLast can be called by any party of the protocol and returns the last (finished) block and its counter: (c, b_c).

For every cryptocurrency there must exist a validation algorithm for validating consistency of the blocks and transactions therein, which we model using the function Extends. It takes as input, a chain of blocks $\mathbf{b}$ and a checkpoint block b_{cp} and outputs 1 if $\mathbf{b} = (b_{\mathsf{cp}+1}, \ldots, b_{\mathsf{cp}+i})$ is a valid chain of blocks extending b_{cp} and otherwise it outputs 0. In Section 6 we give more details on the validation algorithm, and how this function is implemented for the Bitcoin system. Recall, that we assume an adversary which cannot compute a chain of blocks of length k by itself (c.f. Section 4). This guarantees that he cannot produce a false chain such that this function outputs 1. To make the position of some transaction tx inside a chain of blocks explicit, we write $\ell := \mathsf{Pos}(\mathbf{b}, \mathsf{tx})$ when the transaction is part of the ℓ-th block of $\mathbf{b}$. If the transaction is in none of the blocks, the function returns ∞. For more details on the transaction and block verification we refer the reader to [7, 26, 51].

5.2 Modeling the TEE

In order to model the functionality of a TEE, we follow the work of Pass et. al. [54]. We explain here only briefly the simplified version of the TEE functionality whose formal definition can be found in [54, Fig. 1]. On initialization, the TEE generates a pair of signing keys (mpk, msk) which we call master public key and master secret key of the TEE. The TEE functionality has two enclave operations: install and resume. The operation TEE.install takes as input a program p which is then stored under an enclave identifier eid. The program stored inside an enclave can be executed via the second enclave operation TEE.resume which takes as input an enclave identifier eid, a function f and the function input in. The output of TEE.resume is the output out of the program execution and a quote ϱ over the tuple (eid, p, out). In the protocol description we abstract from the details how the users verify the quote that is generated through the enclave attestation. Since we only consider one instance E of the specific program p, we will simplify the resume command $[out, \varrho] := \mathsf{TEE.resume}(eid, \mathsf{f}, in)$ and write[5]:

$$[out, \varrho] := E.\mathsf{f}(in)$$

For every attestable TEE there must exist a function $\mathsf{vrfyQuote}(mpk, p, out, \varrho)$ which on input of a correct quote ϱ outputs 1, if and only if out was outputted by an enclave with master public key mpk and which indeed loaded p. Again, we assume that the adversary cannot forge a quote such that the function vrfyQuote() outputs 1. For more information on how this verification of the attestation is done in practice we refer the reader to [54].

5.3 Detailed Protocol Description

As explained in Section 3, our protocol $\pi_{\text{FastKitten}}$ proceeds in three phases. During the *setup phase* the contract is installed in the enclave, attested, and all parties deposit their

[5] Since we only need the quote of the first activation of E, we will omit this parameter from there on.

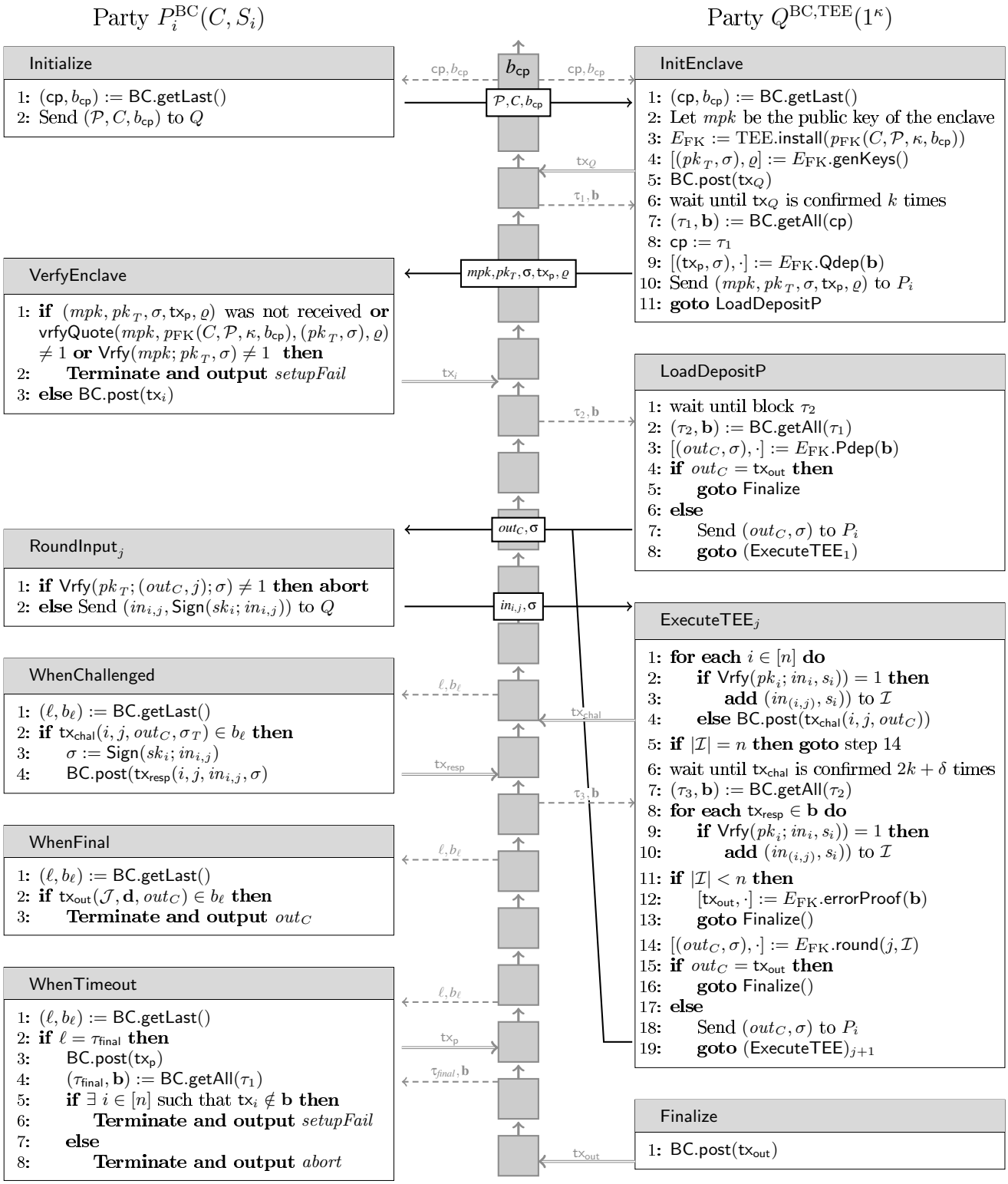

Figure 2: Protocol $\pi_{\text{FASTKITTEN}}$. Direct black arrows indicate communication between the parties and Q, gray dashed arrows indicate reading from the blockchain and gray double arrows posting on the blockchain.

coins. Then the *round execution* follows for all m rounds of the interactive contract. When the contract execution aborts or finishes, the protocol enters the *finalize phase*. We now explain all phases and the detailed protocol steps for all involved parties and the operator Q in depth. The detailed interactions as well as the subprocedure of the parties and the operator are displayed in Figure 2, Figure 3 describes the FASTKITTEN enclave program p_{FK}. Overall the protocol requires six different type of transactions.

Setup phase. In the setup phase, each party P_i first runs Initialize to generate its key pairs and gets the latest block b_{cp} which serves as a genesis block or checkpoint of the protocol. Then P_i sends the set of parties $\mathcal{P}$, the b_{cp} and the contract C to the operator Q. Upon receiving the initial values from all n parties, Q runs the subprocedure InitEnclave to initialize the trusted execution of the enclave program $p_{\mathrm{FK}}(\mathcal{P}, C, \kappa, b_{\mathsf{cp}})$ where κ is the security parameter of the scheme. This security parameter κ also determines the values for the timeout period t and the confirmation constant k. This ensures that all parties and the TEE agree on these fixed values. Once p_{FK} is installed in the enclave, it generates key pairs for the protocol execution and in particular the blockchain public key pk_T[6]. Now, Q can make its deposit transaction tx_Q which assigns q coins to the enclave public key.

Q's Deposit Transaction $\mathsf{tx_Q}$	
tx.Input:	Some unspent tx from Q
tx.Output:	Assign q coins to pk_T

Let block counter τ_1 denote the time when this transaction has been included and confirmed in the blockchain. Q loads all blocks from cp to τ_1 as evidence to the enclave. If this evidence is correct, the execution of p_{FK} function Qdep outputs a penalty transaction $\mathsf{tx_p}$, stating that after timeout τ_{final} (after which the protocol must be terminated) the q coins of Q's deposit transaction tx_Q are payed out to the parties $P_1, \ldots, P_n$.

Penalty Transaction $\mathsf{tx_p}$	
tx.Input:	Q's Deposit Transaction tx_Q
	For all $i \in [n]$:
$\mathsf{tx.Output}_i$:	Assign c_i coins to P_i
tx.Time:	Spendable after τ_{final}

Q sends the penalty transaction to all parties $P_1, \ldots, P_n$, who run subprocedure VerfyEnclave. This transaction is used whenever the protocol does not finish before the final timeout τ_{final}, which equals $(3+2m) \times (\delta + k)$ blocks after the protocol start (recall, that we use δ to bound the time until some transaction is guaranteed to be included and it will be confirmed after k blocks).[7] Only if participant P_i received this penalty transaction from Q during the setup and verified that the program $p_{\mathrm{FK}}(\mathcal{P}, C, \kappa, b_0)$ is installed in the enclave, it creates and publishes its deposit transaction.

$\mathbf{P_i}$'s Deposit Transaction $\mathsf{tx_i}$	
tx.Input:	Some unspent tx from P_i
tx.Output:	Assign c_i coins to TEE

After time $\tau_2 < \tau_1$, Q executes LoadDepositP and again provides the block evidence to the enclave execution of p_{FK}. If all parties published the deposit transactions, the first-round execution starts. Otherwise the enclave proceeds to the finalize phase and outputs a refund transaction $\mathsf{tx}_{\mathsf{out}}(T, \vec{c})$ that returns the deposit back to honest users and Q, where $T \subset \mathcal{P}$ is the set of all parties that submitted the deposit transaction until time τ_2. Note, that the internal state of the contract execution is maintained by the p_{FK} program inside the enclave. This guarantees that the contract is not executed on outdated state.

Round computation phase. When the protocol arrives to the round computation phase, Q sends the authenticated output of the enclave to every party P_i and requests input for the next round. Each party P_i runs the round algorithm. Internally it verifies whether the input request came from the enclave by verifying the attached signature. Then it generates and signs its round input and sends it to Q. While P_i waits for the next round, Q verifies all received inputs and their signatures in the ExecuteTEE subprocedure. If all the parties P_i responded with correctly signed round inputs, Q triggers the execution of the contract in the enclave. Let us emphasize that in this simplified description of our protocol we do not focus on the privacy aspect and hence we omit that all round inputs to the contract could be encrypted with the public key of the enclave. In this case the trusted enclave execution needs to decrypt them before it evaluates the contract on them. See Section 9.3 for more details.

Note that the operator Q may be malicious and refrain from requesting a party P_i for the input to a round computation. Instead Q may pretend that it actually did not receive any input from the party P_i. On the other hand, one can imagine a scenario where Q is behaving honestly but the party P_i is dishonest and does not send the correctly signed round input to Q. Note, that the program p_{FK} cannot distinguish between these two cases without additional information. We will next show how an honest Q can generate a proof to attribute the malicious behavior to P_i. First, Q has to publish a challenge transaction $\mathsf{tx}_{\mathsf{chal}}$ which includes the signed output of the previous step. $\mathsf{tx}_{\mathsf{chal}}$ spends a very small amount μ of coins from Q and assign them to party P_i[8].

[6] For simplicity we omit here, that the enclave might need multiple key pairs for signing transactions and messages.

[7] The definition of τ_{final} guarantees that even if the execution is delayed in every round, an honest operator will not be penalized.

[8] Cryptocurrencies like Bitcoin allow transactions with very small denominations (e.g. fractions of cents).

Challenge Transaction $\mathsf{tx}_{\mathsf{chal}}(i, j, out_C, \sigma_T)$	
tx.Data:	Store i, j, out_C, σ_T
tx.Input:	Some unspent tx from Q
tx.Output:	Assign μ coins to P_i

Once $\mathsf{tx}_{\mathsf{chal}}$ is included in the blockchain, party P_i can read the correct output information from the transaction. The party should respond with $\mathsf{tx}_{\mathsf{resp}}$, which includes its signed round input. $\mathsf{tx}_{\mathsf{resp}}$ spends the $\mathsf{tx}_{\mathsf{chal}}$ and assigns the μ coins back to Q. The action of P_i is depicted via the WhenChallenged subprocedure.

Response Transaction $\mathsf{tx}_{\mathsf{resp}}(\mathbf{i}, \mathbf{j}, in, \sigma_{\mathbf{i}})$	
tx.Data:	Store i, j, in, σ_i
tx.Input:	Challenge Transaction $\mathsf{tx}_{\mathsf{chal}}(i, j, state)$
tx.Output:	Assign μ coins to Q

If some party does not send the response after it was challenged, Q can prove this misbehavior to the FASTKITTEN program, by providing the blockchain evidence of the challenge-response transcript. If the enclave program identifies a cheating party, it proceeds to the finalize phase. Otherwise, if all the parties' inputs were received with authentication (possibly after challenge-response phase), Q instructs the enclave to execute the contract on the accumulated input.
The result of the contract execution is the output out_C, the updated state *state*, and a coin distribution denoted by **d**. If *state* equals $\perp$, the contract execution is finished, and the protocol proceeds to the finalize phase. Otherwise, FASTKITTEN internally stores the state and outputs out_C to Q who sends this output to all parties and waits for next round inputs.

Finalize phase. In the finalize phase, the enclave publishes a final output transaction $\mathsf{tx}_{\mathsf{out}}$ which distributes the coins back to all honest parties. It is parameterized by a set of parties to receive coins $\mathcal{J}$, a final coin distribution $\vec{e}$ and a final state out_C. The transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \vec{e}, out_C)$, spends all deposit transactions tx_i for all $i \in \mathcal{J}$ and Q's deposit transaction tx_Q. It includes the out_C in the data field and assigns q coins back to Q and e_i coins to party P_i, for every $i \in \mathcal{J}$. Let us note that $\mathcal{J} = [n]$ implies correct protocol termination. If $\mathcal{J} \neq [n]$, then some party misbehaved and the protocol failed. Either a party did not make a deposit in the setup phase (signaled by $out_C = setupFail$) or some party aborted in the round computation phase (signaled by $out_C = abort$). In both cases all other parties get their initial deposits back. Note, that if a party P_j is caught cheating by the TEE, it will lose its deposit.
Q now has to publish this transaction to get his coins before time τ_{final} and by that also distributes coins and reveals out_C to honest parties. The participants need to constantly monitor the blockchain for transactions which challenge them or indicate final output. When they see a challenge transaction they respond as described above. If they see an output transaction they know the protocol execution ended and output the final contract output according to subroutine WhenFinal.

Output Transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \vec{e}, out_C)$	
tx.Data:	Store out_C
tx.Input:	Deposit Transactions $\mathsf{tx}_Q, \{\mathsf{tx}_i\}_{i \in \mathcal{J}}$
$\mathsf{tx.Output}_1$:	q coins to Q
	For all $i \in \mathcal{J}$:
$\mathsf{tx.Output}_{i+1}$:	e_i coins to P_i

6 Execution Facility

As shown in Figure 1, we leverage a TEE for smart contract execution. For our prototype, we implemented FASTKITTEN for the Bitcoin blockchain using Intel SGX as a TEE. We chose Python as our scripting engine because it's memory safe, very well known, and widely available. To interact with the Bitcoin blockchain data in the enclave, we implemented our *Crypto library* using the open-source *breadwallet-core* [14], a simplified payment verification (SPV) library for Bitcoin used by the *Breadwallet* mobile wallet app. To abstract from SGX's peculiarities, and thus simplify smart contract development, we use the Graphene Library OS [17] (referred to as "Graphene" in the rest of the paper) as a basis. Graphene enables running arbitrary native Linux binaries in SGX enclaves while providing compatible library interfaces for networking and other OS services. Note that the design of the FASTKITTEN protocol does not require a trusted time source in the TEE.

6.1 The Enclave Program FASTKITTEN

An execution facility in the sense of FASTKITTEN must provide a set of abstract functionalities like key generation, transaction generation, smart contract execution, and error handling, all executed inside the enclave. This set of procedures is described in detail in Figure 3. We implemented each of the procedures using equivalent Python scripts. It is parameterized by the set of parties $\mathcal{P}$, the contract C which internally specifies the expected deposits $\mathbf{c}$, a security parameter κ and a genesis block b_{cp}. This does not need to be the actual genesis block of the underlying blockchain but it can be a later block which is used as a checkpoint. All parties must verify that this block is indeed a block of the blockchain. The security parameter κ also determines the waiting time k which is needed for the verification of the blocks.

6.2 Blockchain Verification

Blockchain communication is important for the setup and the finalization phase in the protocol. Thanks to the integrity properties of blockchains, a secure connection between the enclave and the blockchain is not needed if verification of received data can be done in the enclave. As it is not practical to download a complete copy of the blockchain to the enclave, we only concentrate on transactions caused by FASTKITTEN

The execution of p_{FK} is initialized with the secret key *msk*, the set of parties (where every $P_i \in \mathcal{P}$ is identified by its key pk_i), a contract C, a security parameter κ (which also defines the waiting period t and confirm period k) and a checkpoint b_{cp}. Internally it stores the state of the contract *state* and the status flag s initially set to $state = \emptyset$ and $s = \mathsf{genKeys}$.

procedure genKeys()

1: **if** $s \neq \mathsf{genKeys}$ **then** abort
2: $(sk_T, pk_T) := \mathsf{Gen}(1^\kappa)$
3: $s := \mathsf{Qdep}$
4: **return** $pk_T, \mathsf{Sign}(msk; pk_T)$

procedure Qdep($\mathbf{b}$)

1: **if** $s \neq \mathsf{Qdep}$ **or** $\mathsf{Extends}(b_{\mathsf{cp}}, \mathbf{b}) \neq 1$ **or** $\mathsf{Pos}(\mathbf{b}, \mathsf{tx}_Q) > |\mathbf{b}| - k$ **then** abort
2: $s := \mathsf{Pdep}$
3: $b_{\mathsf{cp}} :=$ last block of $\mathbf{b}$
4: **return** tx_{p} ▷ Else, output penalty transaction

procedure Pdep($\mathbf{b}$)

1: **if** $s \neq \mathsf{Pdep}$ **or** $\mathsf{Extends}(b_{\mathsf{cp}}, \mathbf{b}) \neq 1$ **then** abort
2: set $\mathcal{J} := \emptyset$
3: **for** $i \in \mathcal{P}$ **do**
4: $\quad \ell_i := \mathsf{Pos}(\mathbf{b}, \mathsf{tx}_i)$
5: $\quad$ **if** $\ell_i < \delta$ **and** $\ell_i < |\mathbf{b}| - k$ **then** add i to $\mathcal{J}$
6: **if** $\mathcal{J} = [n]$ **then**
7: $\quad s := \mathsf{round}_1$
8: $\quad b_{\mathsf{cp}} := \mathbf{b}.\mathsf{last}$
9: $\quad$ **return** $\emptyset, \mathsf{Sign}(sk_T; \emptyset, b_{\mathsf{cp}})$
10: **else**
11: $\quad s := \mathsf{terminated}$
12: $\quad$ **return** $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, setupFail)$

procedure round$(j, (in_1, \sigma_1) \ldots, (in_n, \sigma_n))$

1: **if** $s \neq \mathsf{round}_j$ **or** for any $i \in [n] : \mathsf{Vrfy}(pk_i; in_i, s_i) \neq 1$ **then** abort
2: $(out_C, state', \mathbf{d}) := C(state, \vec{in})$
3: **if** $state' \neq \perp$ **then**
4: $\quad s := \mathsf{round}_{j+1}$
5: $\quad state := state'$
6: $\quad$ **return** $out_C, \mathsf{Sign}(sk_T; (out_C, j))$
7: **else**
8: $\quad s := \mathsf{terminated}$
9: $\quad$ **return** $\mathsf{tx}_{\mathsf{out}}([n], \mathbf{d}, out_C)$

procedure errorProof, $(j, \mathbf{b})$

1: **if** $s \neq \mathsf{round}_j$ **or** $\mathsf{Extends}(b_{\mathsf{cp}}, \mathbf{b}) \neq 1$ **then** abort
2: Let $\sigma := \mathsf{Sign}(sk_T; (out_C, j))$
3: $\mathcal{J} := [n]$
4: **for** $i \in \mathcal{P}$ **do**
5: $\quad$ **if** $\mathsf{Pos}(\mathbf{b}, \mathsf{tx}_{\mathsf{chal}}(i, j, out_C, \sigma)) < |\mathbf{b}| - \delta - k$ **then**
6: $\quad\quad$ **if** $\mathsf{Pos}(\mathbf{b}, \mathsf{tx}_{\mathsf{resp}}(i, j, in, \sigma) > |\mathbf{b}| - k$ **then**
7: $\quad\quad\quad$ delete i from $\mathcal{J}$
8: $\quad\quad$ **else if** $\mathsf{Vrfy}(pk_i; in, \sigma) \neq 1$ **then**
9: $\quad\quad\quad$ delete i from $\mathcal{J}$
10: $s = \mathsf{terminated}$
11: **if** $\mathcal{J} \neq [n]$ **then**
12: $\quad$ **return** $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, abort)$

Figure 3: FASTKITTEN enclave program $p_{\mathrm{FK}}(\mathcal{P}, C, \kappa, b_{\mathsf{cp}})$

protocol invocation. Thus, it is sufficient to verify that these transactions are part of a valid block—without downloading entire blocks, which can be done efficiently using simplified payment verification (SPV). However, SPV libraries can only prove that a transaction *is* part of a block on the blockchain, but they cannot prove that a transaction *is not* part of any block. As required by the challenge-response case, we added an alternative verification mode that fully downloads every block that could potentially contain the transaction and checks whether its present in any of those blocks.

6.3 Participant Communication

To place the deposits and receive them later, as well for sending input, communication between participants (including the Operator Q) is needed in the off-chain phase. We secure this communication using TLS sockets provided by Python. This transparently encrypts participants' communication, and thus ensures input integrity and confidentiality of parties' messages towards the operator.

6.4 Enclave Setup

In the FASTKITTEN prototype, we leverage Intel SGX as a TEE. SGX is a TEE included in recent Intel CPUs which introduces the concept of isolated hardware *enclaves* that can be created and managed using new CPU instructions. SGX enclaves are even shielded from the operating system; only the CPU is trusted. To support smart contract execution in these enclaves we provide a run-time environment based on Graphene, which replaces the Intel SDK in both the enclave and the host process. This allows Graphene to transparently provide services from the untrusted OS (and check the integrity of the results). To protect the enclave application from the host process, a *manifest* has to be provided at enclave initialization. The manifest includes interfaces, services, and respective integrity checksums, e.g., hashes of files the enclave requires. Accesses to these files will be checked against hashes in the manifest to guarantee integrity.

As depicted by Figure 3, the Execution Facility incorporates a set of functionalities. For key derivation (*genKeys*) we leverage the `rdrand` instruction to get high-entropy randomness inside of the enclave. After checking that tx_Q (*Qdep*) is in the blockchain, the derived private key sk_T is used to generate the penalty transaction tx_p using our Crypto library. tx_p is distributed to the other participants over a TLS connection. Other participants can generate their deposit transactions tx_i (*Pdep*) using a regular wallet. This concludes the setup phase, and the smart contract gets executed (*round*).

The Graphene run-time environment enables FASTKITTEN to support arbitrary Linux binaries, thus, can be used to implement smart contracts. However, instead of allowing binaries, we use a scripting engine based on a Python interpreter in our proof-of-concept implementation. First, this makes development easier for contract developers, as they are not always familiar with lower-level programming languages, and second,

this makes smart contracts less prone to memory corruption vulnerabilities. Two use cases we implemented are presented and evaluated in Section 8.

6.5 Denial of Service Protection

The protocol as described in Section 5 assumes instantaneous contract execution meaning that the execution of a contract inside a TEE takes no time. For most practical contracts, this simplifying assumption is reasonable since executing a simple contract function inside a TEE is much faster than the network/blockchain delay. However, this is not true when considering arbitrary contracts which might potentially contain endless loops. Moreover, the halting problem states that it is impossible to predict if a certain algorithm will halt within a certain number of steps. A simple protection against endless loops and denial-of-service attacks, is letting the enclave monitor the execution of the smart contract and terminate execution if the number of execution steps exceeds a predefined limit. If the contract execution is aborted due to an execution timeout, the enclave signs an outputs transaction $\mathsf{tx}_{\mathsf{out}}$ which returns deposited coins back to parties and to the operator.

7 Security

In this section we present the underlying security considerations of FASTKITTEN.

7.1 Protocol Security

Due to limited space, we present our novel model in the extended version of this paper, where we also formally state the security properties, the formal statement of the theorem as well as the proof. Here we will only briefly explain the security properties.

In order to guarantee security for the protocol, we require three security properties: *correctness*, *fairness* and *operator balance security*.

Intuitively, correctness states that in case all parties behave honestly (including the operator), every party $P_i \in \mathcal{P}$ outputs the correct result and earns the amount of coins she is supposed to get according to the correct contract execution. The fairness property guarantees that if at least one party $P_i \in \mathcal{P}$ is honest, then (i) either the protocol correctly completes an execution of the contract or (ii) all honest parties output *setupFail* and stay financially neutral or (iii) all honest parties output *abort*, stay financially neutral, and at least one corrupt party must have been financially punished. Finally, the operator balance security property says that in case the operator behaves honestly, he cannot lose money.

Theorem 1 (Informal statement)**.** *The protocol* $\pi_{\text{FASTKITTEN}}$ *as defined in Section 5 satisfies correctness, fairness and operator balance security property.*

The most challenging part of the proof is the fairness property. We need to show how honest parties reach consensus on the result of the execution and prove that coins are always distributed between parties according to this result (even if malicious parties collude with the operator). In order to prove the operator balance security, we show that an honest operator has always enough time to publish a valid output transaction which pays him back his deposit, before the time-locked penalty transaction can be posted on the blockchain.

Incentive-driven adversary If we consider only incentive-driven adversaries, then statement (iii) of the fairness property is never true. Hence, if the setup phase completes successfully, then the result of the protocol is a correct contract execution. This follows directly from the fact, that when the protocol aborts the misbehaving parties lose coins. By definition of incentive-driven parties, losing coins is against their interest. This is why the only possible outcome of the protocol is correct execution of the contract. Moreover, when we consider fees for positing transaction on the blockchain, parties are additionally incentivized to prevent the challenge-response transactions. These additional incentives enforce fast and protocol compliant behavior of the parties.

7.2 Architecture Security

The main goal of FASTKITTEN is to enable efficient execution of general multi-round smart contracts. Hence, we analyze the security of FASTKITTEN with regards to its system architecture and implementation. Possible adversaries can be malicious participants, a malicious operator, or a combination of both.

We note that participating clients are only required to send and receive transactions from the blockchain (e.g., to enter an execution) and the ability to exchange protocol messages (e.g., to play rounds). Hence, client implementations can be based on a diverse set of entirely different code bases in practice, possibly using memory-safe languages such as Python, Go, or Rust. Malicious participants are further limited to interacting with other parties and the operator through the exchange of messages as specified within our protocol, and hence, we focus on the TEE-based execution facility in the following.

A malicious operator could deny execution, however, he is incentivized to adhere to the protocol or lose money. Thus, we assume that the goal of a malicious operator is to try and exploit the execution facility at runtime. Since the operator already controls the host process, the main target would be the enclave that executes the contract. Enclaves have a well-defined interface with the rest of the system, and any attack has to be launched using this interface. By providing fake data through this interface, the attacker could try to exploit a memory-corruption vulnerability in the low-level enclave code to launch (a) a code-reuse attack, e.g., by manipulating enclave stack memory, or (b) a data-only attack, e.g., to leak information about the game state or manipulate Bitcoin addresses in contracts. As mentioned in Section 4, for (a) we assume a standard code-reuse defense such as control-flow integrity [3, 15, 50, 62, 65] or fine-grained code randomization [21,23,30,42,53,61]. The core functionality of

FASTKITTEN additionally tackles both attack vectors by implementing the main enclave code in Python, which provides memory-safety features such as implicit bounds checking. The only parts that are implemented in unsafe languages are the initialization code of Graphene [17] and the Simple Payment Verification (SPV) library [14]. FASTKITTEN actually has no strong dependency on Graphene in principle, it was mainly used to simplify and speed up prototype implementation. Finally, SPV represents a standard library used by most blockchain clients and an adversary that is able to construct a data-only attack against it would be able to exploit any of those clients connected to the Bitcoin network using the same data-only attack.

8 FASTKITTEN Contracts

In this section we take a look at applications and performance through a number of benchmarks.

8.1 Complexity

The FASTKITTEN protocol consists of *setup*, *round computation* and *finalize* phases. During the *setup* phase, each party P_i deposits a constant amount of coins c_i. The operator needs to deposit an amount $\sum_{i\in[n]} c_i$ which equals the sum of all other deposits from $\mathcal{P}$ together. To post the deposit transactions tx_is and tx_Q, a total of $n+1$ transactions is necessary.
During the *round computation phase*, in the optimistic case FASTKITTEN can operate completely off-chain without any blockchain interaction. Any user can force that challenge response transactions are posted to attribute misbehavior of a party, in any given round. If this (pessimistic) case occurs, it can add 2 to another $2n$ transactions. In the worst case, a challenge response transaction pair needs to be posted on the blockchain for every party P_i at every round $j \in [m]$ leading to $\mathcal{O}(nm)$ blockchain interactions. In *finalize* phase, FASTKITTEN requires one additional payout transaction $\mathsf{tx}_{\mathsf{out}}$ to settle money distribution among parties. Scenarios of missing deposit at the *Setup* phase or an abort by a party at the *round computation* phase are dealt with by posting the refund transaction $\mathsf{tx}_{\mathsf{out}}$ and the penalty transaction tx_{p} respectively.

Setup time In the optimistic case (which we have shown is the standard case when considering incentive-driven parties) the overall execution of the protocol only requires $n+2$ transactions on the blockchain. This also indicates at what speed the protocol can be executed in this case. If all parties agree, the setup phase can be finished in 2 blockchain rounds and from that point on the protocol can be played off-chain. In the next subsection we give some indication how fast this second part can be achieved. Running the protocol as fast as possible is in the interest of every party since it shortens the locking time of the deposits.

8.2 Performance Evaluation

We performed a number of performance measurements to demonstrate the practicality of FASTKITTEN using our lab setup, which consists of three machines: First, an SGX-enabled machine running Ubuntu 16.04.5 LTS with an Intel i7-7700 CPU clocked at 3.60GHz and 8GB RAM, where we installed FASTKITTEN's contract execution facility to play the role of the operator's server. Second, a machine running Ubuntu 14.04.4 LTS on an Intel i7-6700 CPU clocked at 3.40GHz with 32GB RAM, which provides unmodified blockchain nodes in a local test network using Bitcoin Core version 0.16.1. Third, a laptop machine with macOS 10.13.6 on with Intel i7-4850HQ CPU clocked at 2.30GHz and 16GB of RAM, which takes the role of the participants in the protocol. All three machines are connected through a Gigabit Ethernet LAN. For tests involving the real Bitcoin network the individual machines are connected through the Internet using our Internet connection.

Block validation In our experiments, the enclave takes approximately 5 s to validate one block from the Bitcoin main network, thus proving that it is capable of validating real blocks in real time.

Enclave Startup The time to setup an enclave until it is ready is 2 s, proving that instantiating enclaves on the fly is feasible.

End-to-end Time Assuming all parties are incentive-driven and, thus, comply with the protocol, the total time required by FASTKITTEN is the time of 2 blockchain interactions (see Section 8.1), plus the computation time (a few milliseconds in our use cases), plus the time required by the parties to choose the next inputs.

8.3 Applications

FASTKITTEN allows to run complex smart contracts on top of cryptocurrencies that would not natively support such contracts, like Bitcoin. But in contrast to Turing-complete contract execution platforms like Ethereum, a secure off-chain execution such as FASTKITTEN puts some restrictions on the contracts it can run:

- The number of parties interacting with the contract must be known at the start of the protocol.
- It must be possible to estimate an upper bound on the number of rounds and the maximum run time of any round.

All of these restrictions make FASTKITTEN contracts different from smart contracts running on Ethereum itself. The restrictions above come from the fact that the contract can be completely (and repeatedly) executed without blockchain interactions. Other off-chain solutions (like state channels [20,24,49]) come with similar caveats. By allowing additional blockchain interaction we could get around those restrictions but we would lose efficiency in the optimistic case (which is also similar to state channel constructions).
FASTKITTEN has important features which are supported by neither Bitcoin nor Ethereum — FASTKITTEN allows private inputs and batched execution of user inputs. Overall, this leads to cheaper, faster and private contract execution than what

is possible with on-chain contracts in Ethereum. Below, we highlight these efficiency gains by presenting four concrete use-cases in which FASTKITTEN outperforms contracts run over Ethereum or in Ethereum state channels.

Lottery A lottery contract takes coins from every involved party as input, and randomly selects one winner, who gets all the coins. The key challenge for such a contract is to fairly generate randomness to select the winner. In Ethereum or Bitcoin the randomness is computed from user inputs through an expensive commit-reveal scheme [48]. In FASTKITTEN, all parties can immediately send their random inputs to the enclave which will securely determine a winner. Hence, we reduce the round complexity from $\mathcal{O}(\log n)$ [48] to $\mathcal{O}(1)$.

Auctions Another interesting use-case for smart contracts are auctions, where parties place bids on how much they are willing to pay and the contract determines the final price. In a straightforward auction, the bids can be public, but more fair versions, like second bid auctions, require the users not to learn the other bids before they place their own. The privacy features of FASTKITTEN can be used to reduce the round complexity for such auctions which would otherwise require complex cryptographic protocols [25].

Rock-paper-scissors We implemented the popular two-party game rock-paper-scissors to show the feasibility of FASTKITTEN contracts. Again, the privacy features allow one match to be executed in a single round, which would have required at least 3 rounds in Ethereum. The pure execution time in the optimistic case, excluding delays due to human reaction times, is 12ms for one round (averaged over 100 matches). This demonstrates that off-chain protocols, like FASTKITTEN, are highly efficient when the same set of parties wants to run complex contracts (like multiple matches of a game).

Poker We also implemented a Texas Hold'em Poker game, to prove that multi-party contracts which inherently require multiple rounds can also be efficiently executed in FASTKITTEN. In our implementation, each player starts with an equal chip stack and participates in an initial betting round and in additional rounds after the flop, river, and turn have been dealt by the enclave. If more than two players remain in the game after the final bets, the enclave reveals the winner and distributes the chips in the current pot to the winner. The game continues until only one player remains. We measured 50 matches between 10 players resulting in an average time of 45ms per match (multiple betting rounds are included in each match). The run time was measured starting from the moment all deposits are committed to the blockchain.

Real-world Fees We generated examples of the transaction types used in our protocol for a 10-player poker match. In Table 2 we estimate the fees required to commit to the blockchain our transactions, in addition to a typical deposit transaction. Assuming all parties comply with the protocol, each party (including Q) must pay between 0.05 USD and 0.46 USD for the deposit. Additionally, the output transaction $\mathsf{tx}_{\mathsf{out}}$ requires between 0.36 USD and 3.65 USD in fees.

Transaction	Size (Bytes)	Fees (BTC)	Fees (USD)
Deposit (typical)	250	0.000007-0.000073	0.05-0.46
Penalty (tx_{p})	504	0.000015-0.000148	0.09-0.93
Challenge ($\mathsf{tx}_{\mathsf{chal}}$)	293	0.000009-0.000086	0.05-0.54
Response ($\mathsf{tx}_{\mathsf{resp}}$)	266	0.000008-0.000078	0.05-0.49
Output ($\mathsf{tx}_{\mathsf{out}}$)	1986	0.000058-0.000582	0.36-3.65

Table 2: Estimated fees for a typical deposit transaction and the FASTKITTEN transactions, using data from CoinMarketCap [2] and BlockCypher [1] retrieved on Nov. 14, 2018.

Other Well-known Contracts Certain well-known contracts like ERC20 token and CryptoKitties inherently need to be publicly available on the blockchain, since they are accessed frequently by participants which are not previously known. In contrast, contracts resembling our examples above, which rely on private data and where a fixed set of participants sends a large number of transactions, are highly efficient when moved off-chain using a system like FASTKITTEN. The nature of off-chain solutions like FASTKITTEN or state channels requires advance knowledge of the participants. Open contracts like ERC20 and CryptoKitties that require continuous synchronization with the blockchain and are meant to be publicly accessible would eliminate the advantages of off-chain solutions.

9 Discussion and Extensions

In order to explain and analyze the FASTKITTEN protocol, we presented a simplified protocol version which only includes the building blocks required to guarantee security. Depending on the use case one might be interested in further properties. Possible extensions discussed in this section include the option to pay the operator for his service, protect the operator against TEE faults, hide the contract output from through a layer of output encryption and allow cross-currency smart contracts. In the following, we explain how to achieve these features and at what cost they can be added to the simplified protocol.

9.1 Fees for the Operator

The owner of the TEE provides a service to the users who want to run a smart contract and, naturally, he wants to be paid for it. In addition to the costs of buying, maintaining and running the trusted hardware, he also needs to block the security deposit q for the duration of the protocol. While the security of FASTKITTEN ensures that he will never lose this money, he still cannot use it for other purposes. The goal of the operator-fees is to make both investments attractive for Q. We assume that the operator will be paid ξ coins for each protocol round for each party. Since the maximum number of rounds m is fixed at the protocol start, Q will receive $\xi \times$

$n \times m$ coins if the protocol succeeds (even if the contract terminated in less than m rounds). If the operator proves to the TEE in round x that another party did not respond to the round challenge, he will only receive a fee for the passed x number of rounds (namely $\xi \times x \times n$). This pay-per-round model ensures that the operator does not have any incentive to end the protocol too early. If the protocol setup does not succeed or the operator cheats, he will not receive any coins. The extended protocol with operator fees requires each party to lock $c_i + m \times \xi$ coins and the operator needs to level this investment with $qc_i + m \times \xi$ coins.

9.2 Fault Tolerance

In order to ensure that the execution of the smart contract can proceed even in the presence of software or hardware faults, the enclave can save a snapshot of the current state in an encrypted format, e.g., after every round of inputs. This encrypted state would be sent to the operator and stored on redundant storage. If the enclave fails, the operator can instantiate a new enclave which will restart the computation starting from the encrypted snapshot. If the TEE uses SGX, snapshots would leverage SGX's sealing functionality [31] to protect the data from the operator while making it available to future enclave instances.

9.3 Privacy

As mentioned in the introduction, traditional smart contracts cannot preserve privacy of user inputs and thus always leak internal data to the public. In contrast to common smart contract technologies, the FASTKITTEN protocol supports privacy preserving smart contracts as proposed in Hawk [36]. This requires *private contract state* to hide the internal execution of the contract and *input privacy*, which means that no party (including the operator) sees any other parties' round input before sending its own.

It is straightforward to see that FASTKITTEN has a secret state, since it is stored and maintained inside the enclave. Input privacy can easily be achieved by encrypting all inputs with the public key of the enclave. This guarantees that only the FASTKITTEN execution facility and the party itself knows the inputs. If required, FASTKITTEN could also be extended to support privacy of outputs from the contract to the parties, by letting the enclave encrypt the individual outputs with the parties' public keys. But this additional layer should only be used when the contract requires it, since in the worst case this increases the output complexity of the challenge and output transaction.

9.4 Multi-currency Contracts

FASTKITTEN requires from the underlying blockchain technology that transactions can contain additional data and can be timelocked. Any blockchain like Bitcoin, Ethereum, Litecoin and many others which allow these transaction types can be used for the FASTKITTEN protocol. With some minor modifications FASTKITTEN can even support contracts which can be funded via multiple different currencies. This allows parties that own coins in different currencies to still execute a contract (play a game) together. The main modification to the FASTKITTEN protocol is that the operator and the enclave need to simultaneously handle multiple blockchains in parallel. In particular, for each of the considered currencies, Q needs to deposit the sum of all coins that were deposited by parties in that currency. This is in order to guarantee that if the operator cheats, players get back their invested coins in the correct currency. In addition, the operator is obliged to challenge each party via its blockchain. If the execution completes (or the operator proves to the enclave that one of the players cheated), the enclave signs one output transaction for each of the currencies. While this extension adds complexity to the enclave program and leads to more transactions and thus transaction-fees, the overall deposit amount stays identical to the single blockchain use case.[9] A complete design and proof of correctness of a cross-ledger FASTKITTEN is left to future work.

10 Conclusion

In this paper we have shown that efficient smart contracts are possible using only standard transactions by combining blockchain technology with trusted hardware. We present FASTKITTEN, our Bitcoin-based smart contract execution framework that can be executed off-chain. Since FASTKITTEN is the first work that supports efficient multi-round contracts handling coins, for the first time, this enables real-time application scenarios, like interactive online gaming, with millisecond round latencies between participants. We formally prove and thoroughly analyze the security of our general framework, also extensively evaluating its performance in a number of use cases and benchmarks.

Additionally, we discuss multiple extensions to our protocol, such as adding output privacy or operator fees, which enrich the set of features provided by our system.

Acknowledgments

We are grateful to our anonymous reviewers and our shepherd Mihai Christodorescu for their constructive feedback.

This work has been supported by the German Research Foundation (DFG) as part of projects HWSec, P3 and S7 within the CRC 1119 CROSSING and the Emmy Noether Program FA 1320/1-1, by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, by BMBF within the iBlockchain project, by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

[9] This solution assumes that any party can receive coins in any of the considered currencies.

Availability

An extended version of this paper, which includes the bytecode of our sample Bitcoin transactions, will be publicly available at the Cryptology ePrint Archive at https://eprint.iacr.org.

References

[1] BlockCypher, Nov 2018. https://live.blockcypher.com/btc/.

[2] CoinMarketCap, Nov 14 2018. https://coinmarketcap.com.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13, 2009.

[4] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[5] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[6] ARM Limited. Security technology: building a secure system using TrustZone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2008.

[7] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO*, 2017.

[8] J. Barbie. Why smart contracts are not feasible on plasma, Jul 2018. https://ethresear.ch/t/why-smart-contracts-are-not-feasible-on-plasma/2598.

[9] G. Belisle. A glimpse into the future of blockchain, 2018. Available at https://the-blockchain-journal.com/2018/03/29/a-glimpse-into-the-future-of-blockchain/.

[10] I. Bentov, Y. Ji, F. Zhang, Y. Li, X. Zhao, L. Breidenbach, P. Daian, and A. Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. *IACR Cryptology ePrint Archive*, 2017.

[11] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel sgx. In *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 2018.

[12] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A. Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, 2017.

[13] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies*, 2017.

[14] Breadwallet. Breadwallet-core - spv bitcoin c library, 2018.

[15] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-flow integrity: Precision, security, and performance. *CoRR*, 2016.

[16] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[17] C. che Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference*, 2017.

[18] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *ACM Symposium on Information, Computer and Communications Security*, 2017.

[19] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. *arXiv preprint arXiv:1804.05141*, 2018.

[20] J. Coleman, L. Horne, and L. Xuanji. Counterfactual: Generalized state channels, Jun 2018. https://l4.ventures/papers/statechannels.pdf.

[21] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies*, 2016.

[22] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, 2016.

[23] L. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2013.

[24] S. Dziembowski, S. Faust, and K. Hostáková. General state channel networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018.

[25] H. Galal and A. Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In *International Conference on Financial Cryptography and Data Security, Trusted Smart Contracts Workshop. Springer*, 2018.

[26] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In *CRYPTO*. Springer, 2017.

[27] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium*, 2017.

[28] M. Hachman. Intel's plan to fix meltdown in silicon raises more questions than answers. https://www.pcworld.com/article/3251171/components-processors/intels-plan-to-fix-meltdown-in-silicon-raises-more-questions-than-answers.html, 2018.

[29] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[30] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2013.

[31] Intel. Intel Software Guard Extensions developer guide, 2016. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf.

[32] Intel. Resources and Response to Side Channel L1 Terminal Fault. https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html, 2018.

[33] A. Juels, A. E. Kosba, and E. Shi. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016.

[34] H. A. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. Arbitrum: Scalable, private smart contracts. In *USENIX Security Symposium*, 2018.

[35] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.

[36] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy*, 2016.

[37] R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[38] R. Kumaresan and I. Bentov. Amortizing secure computation with penalties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[39] R. Kumaresan, T. Moran, and I. Bentov. How to use bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[40] R. Kumaresan, V. Vaikuntanathan, and P. N. Vasudevan. Improvements to secure computation with penalties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[41] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1982.

[42] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[43] J. Lind, O. Naor, I. Eyal, F. Kelbert, P. R. Pietzuch, and E. G. Sirer. Teechain: Reducing storage costs on the blockchain with offline payment channels. In *11th ACM International Systems and Storage Conference*, 2018.

[44] R. Matzutt, J. Hiller, M. Henze, J. H. Ziegeldorf, D. Müllmann, O. Hohlfeld, and K. Wehrle. A quantitative analysis of the impact of arbitrary blockchain content on bitcoin. In *Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC). Springer*, 2018.

[45] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[46] Microsoft. The coco framework, 2018. GIT repository available at `https://github.com/Azure/coco-framework`.

[47] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013.

[48] A. Miller and I. Bentov. Zero-collateral lotteries in bitcoin and ethereum. In *Security and Privacy Workshops (EuroS&PW), 2017 IEEE European Symposium on*. IEEE, 2017.

[49] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry. Sprites: Payment channels that go faster than lightning. *CoRR*, abs/1702.05812, 2017.

[50] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS*, 2015.

[51] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008.

[52] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security symposium*, USENIX Sec, 2013.

[53] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

[54] R. Pass, E. Shi, and F. Tramèr. Formal abstractions for attested execution secure processors. *IACR Cryptology ePrint Archive*, 2016.

[55] J. Poon and V. Buterin. Plasma: Scalable autonomous smart contracts, Aug 2017. Plasma, `https://plasma.io/plasma.pdf/`.

[56] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Annual Network and Distributed System Security Symposium*, 2017.

[57] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Annual Network and Distributed System Security Symposium*, 2017.

[58] J. Teutsch and C. Reitwießner. A scalable verification solution for blockchains, Nov 2017. `https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf`.

[59] F. Tramèr, F. Zhang, H. Lin, J. Hubaux, A. Juels, and E. Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P*, 2017.

[60] J. Van Bulck, F. Piessens, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[61] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2012.

[62] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[63] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016.

[64] F. Zhang, P. Daian, I. Bentov, and A. Juels. Paralysis proofs: Safe access-structure updates for cryptocurrencies and more. *IACR Cryptology ePrint Archive*, 2018.

[65] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*, USENIX Sec, 2013.

A Further Related Work

There is a large body of work trying to improve the scalability of blockchains by moving a major part of smart contract executions off the blockchain (for example, via second layer solutions [24, 34, 49, 55] or outsourcing of computation [58]). As discussed in the main body of this paper, all of these solutions run on top of blockchains with sufficiently complex scripting language, e.g., on Ethereum. However, they cannot be integrated into popular legacy cryptocurrencies such as Bitcoin, which is their main difference compared to our work. Recall that one of the main goals of FASTKITTEN is make minimal assumption on the underlying blockchain technology and in particular, to run over the Bitcoin blockchain.

Another motivation for off-chain contract execution might be the goal of protecting privacy. Hawk [36] and the "Ring of Gyges" [33] are examples of works that do keep the state, all inputs and outputs private. It is also true for the scaling solutions mentioned above; These techniques work only over cryptocurrencies with support for complex smart contracts, e.g. over Ethereum.

Below we discuss the differences between these solutions and FASTKITTEN when run on top of Ethereum.

A.1 Second-layer Scaling Solutions

State Channels State channels [20, 24, 49] are a prominent second layer scaling solution. They allow a set of parties

to execute complex smart contracts off-chain. As long as all parties are honest and agree on the state transitions, the blockchain is contacted only during the channel creation, when parties lock funds in the channel, and during channel closure, when the locked funds are distributed back to the parties according to the result of contract execution. However, once parties run into disagreement off-chain, they have to resolve their dispute on-chain and perform the state transition via the blockchain.

While in the optimistic case when all parties are honest, state channels are very efficient, a potentially heavy computation might need to be done on-chain in case of disagreement. This is in contrast to the FASTKITTEN protocol which does not require any computation to be performed on the blockchain even in case of disputes.

Plasma Another promising second-layer scaling solution is Plasma, first introduced by Poon and Buterin [55]. The main idea of Plasma is to build new chains (Plasma chains) on top of the Ethereum blockchain. Each Plasma chain has its own operator that is responsible for validating transactions and regularly posting a short commitment about the current state of the Plasma chain to a smart contract on the Ethereum blockchain. The regular commitments guarantee to the participants of the Plasma chain that in case the operator cheats, his misbehavior can be proven to the Ethereum smart contract and parties can exit the Plasma chain with all their funds.

While the original goal of Plasma [55] was to support arbitrary complex smart contracts, to the best of our knowledge, there is no concrete protocol that would achieve this goal (the existing Plasma designs support only payment transactions). Moreover, the plasma research community currently conjectures that Plasma with general smart contracts might be impossible to construct [8].

A.2 Incentive-driven Verification

Arbitrum The disadvantage of state channels, i.e., the potentially heavy on-chain execution in case of dispute, is being addressed by the work Arbitrum [34]. Every smart contract, which Arbitrum models as a virtual machine (VM), to be executed off-chain has a set of "manager" parties responsible for correct VM execution. As long as managers reach consensus on the VM state transitions, execution progresses off-chain similarly as in state channels. In case of dispute, managers do not perform the VM state transition on-chain as in state channel. Instead, one manager can propose the next VM state which other managers can challenge. If the newly posted state is challenged, the proposer and the challenger run an interactive protocol via the blockchain, so-called "bisection" protocol, in which one disputable computation step is eventually identified and whose correct execution is verified on-chain. Hence, instead of executing the entire state transition on-chain (which might potentially require a lot of time/space), only one computation step of the state transition has to be performed on-chain in addition to the bisection protocol (which might require $\mathcal{O}(\log(s))$ blockchain transactions, where s is the number of computations steps in the state transition). The Arbitrum protocol works under the assumption that at least one manager of the VM is honest and challenges false states if they are posted by other managers. Since the blockchain interaction during the bisection protocol is rather expensive, Arbitrum uses monetary incentives to motivate managers to behave honestly and follow the protocol.

TrueBit Another solution that supports off-chain execution of smart contracts using incentive verification is TrueBit [58]. For each off-chain execution, the TrueBit system selects (using a lottery) one party, called the "Solver", that is responsible for performing the state transition and inform all other parties about the new contract state. The TrueBit system incentives parties to become so called "verifiers" and check the correctness of the computation performed by the Solver. In case they detect misbehavior, they are supposed to challenge the Solver on the blockchain and run the "verification game" which works similarly as the "bisection protocol" of Arbitrum. Similar to Arbitrum, TrueBit relies on the assumption that there is at least one honest verifier which correctly performs all the validations and challenges malicious Solvers. In contrast to Arbitrum, all inputs and the contract state are inherently public even in the optimistic case when everyone is honest.

Apart from the different trust model and lower requirement on the underlying blockchain technology, FASTKITTEN differs from Arbitrum and TrueBit by providing stronger privacy guarantees, meaning that in both the optimistic and the pessimistic case, inputs of honest parties as well as the state of the smart contract remains private.

A.3 TEEs for privacy

None of the solutions discussed above achieves privacy preserving off-chain contract execution. This is tackled by the work Hawk [36] which keeps the state, all inputs and all outputs private. Hawk contracts [35] achieve these properties using Ethereum smart contracts that judge computations done by a third party (a manager), who executes the contract on private inputs and is trusted not to reveal any secrets. First all parties submit their encrypted inputs to the contract, then the manager computes the result and proves its correctness with a zero knowledge proof. If the proof is correct, the contract pays out money accordingly. While the authors of Hawk discuss the possibility to use SGX for instantiating the manager and reducing the trust assumptions in this party, it still leverages the blockchain for every user input, and it only supports single round protocols which is their main difference to FASTKITTEN. A possible extension to multi-round protocols would be difficult to achieve without letting the smart contract verify the correctness of every round individually, and thus create a large blockchain communication overhead.

HYBCACHE: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments

Ghada Dessouky, Tommaso Frassetto, Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
{ghada.dessouky, tommaso.frassetto, ahmad.sadeghi}@trust.tu-darmstadt.de

Abstract

Modern multi-core processors share cache resources for maximum cache utilization and performance gains. However, this leaves the cache vulnerable to side-channel attacks, where inherent timing differences in shared cache behavior are exploited to infer information on the victim's execution patterns, ultimately leaking private information such as a secret key. The root cause for these attacks is mutually distrusting processes sharing the cache entries and accessing them in a deterministic and consistent manner. Various defenses against cache side-channel attacks have been proposed. However, they suffer from serious shortcomings: they either degrade performance significantly, impose impractical restrictions, or can only defeat certain classes of these attacks. More importantly, they assume that side-channel-resilient caches are required for the entire execution workload and do not allow the possibility to selectively enable the mitigation only for the security-critical portion of the workload.

We present a generic mechanism for a flexible and soft partitioning of set-associative caches and propose a hybrid cache architecture, called HYBCACHE. HYBCACHE can be configured to selectively apply side-channel-resilient cache behavior only for isolated execution domains, while providing the non-isolated execution with conventional cache behavior, capacity and performance. An isolation domain can include one or more processes, specific portions of code, or a Trusted Execution Environment (e.g., SGX or TrustZone). We show that, with minimal hardware modifications and kernel support, HYBCACHE can provide side-channel-resilient cache *only* for isolated execution with a performance overhead of 3.5–5%, while incurring no performance overhead for the remaining execution workload. We provide a simulator-based and hardware implementation of HYBCACHE to evaluate the performance and area overheads, and show how HYBCACHE mitigates typical access-based and contention-based cache attacks.

1 Introduction

For decades now, upcoming processor generations are being augmented with novel performance-enhancing capabilities. Performance and security of processor architectures and microarchitectures are considered exclusively independent design metrics, with architects primarily focused on the more tangible performance benefits. However, the recent outbreak of micro-architectural cross-layer attacks [4–6, 18, 19, 22, 42, 44, 46, 47, 50, 56, 59, 68, 70, 79], has demonstrated the critical and long-ignored effects of micro-architectural performance optimizations on systems from a security standpoint. It is becoming evident how performance and security are at conflict with each other unless architects address the design trade-off early on and not as an afterthought.

One prominent performance feature and the subject of a wide range of recent architectural attacks is the use of caches and cache-like structures to provide orders-of-magnitude faster memory accesses. The intrinsic timing difference between a cache hit and miss is one of various *side channels* that can be exploited by an adversary process via a carefully crafted side-channel attack to infer the memory access patterns of a victim process [23,25–29,34,35,38,54,61,71,77,78]. Consequently, the adversary can leak unauthorized information, such as a private key, hence violating the confidentiality and isolation of the victim process.

Cache Side-Channel Attacks. In earlier years, cache side-channel attacks have been shown to compromise cryptographic implementations [8, 54, 61, 78]. More recently, attack variants such as *Prime + Probe* [34, 38, 54, 61] and *Flush + Reload* attacks [29, 78] are being demonstrated on a much larger scale. They have been shown to bypass address space layout randomization (ASLR) [23, 25], infer keystroke behavior [26, 27], or leak privacy-sensitive human genome indexing computation [11], whereby millions of platforms using various architectures have been shown vulnerable to such attacks. The attacks require an adversary to orchestrate particular cache evictions of target memory addresses of interest and

after a time interval measure its own memory access latencies or observe relevant computation and profile how it has been affected. This enables the adversary to deduce the victim's memory access patterns and infer dependent secrets. Cache side-channel attacks have been shown to exploit core-specific caches as well as shared last-level caches across different cores or virtual machines [27,38,54]. Even hardware-security extensions and trusted execution environments (TEEs) such as Intel SGX [13, 33] and ARM TrustZone [7] are not immune to these attacks. While they do not claim cache side-channel security, recent cache side-channel attacks targeting SGX [11,21,60,66] and TrustZone [49,80] have been shown to compromise the acclaimed privacy and isolation guarantees of these security architectures, thus undermining their very purpose.

Existing Cache Defenses. To defeat cache side-channel attacks, there has been extensive research on techniques to identify and mitigate information leaks in a software's memory access patterns [16, 17, 45]. However, mitigating these leaks efficiently for arbitrary software (beyond cryptographic implementations) remains impractical and challenging. Alternatively, hardware-based and software approaches have been proposed to modify the cache organization itself to limit cache interference across different security domains. Examples include modifying replacement and leveraging inclusion policies [39,76], as well as approaches that rely on cache partitioning [24,40,41,51,72,73,82], and randomization/obfuscation-based schemes [52,53,63,69,73] to randomize the relation between the memory address and its cache set index.

While strict cache partitioning is the intuitive approach to provide complete cache isolation and non-interference between mutually distrusting processes, it remains highly impractical and prevents efficient cache utilization. On the other hand, randomization-based approaches make the attacks computationally much more difficult by randomizing the mapping of memory addresses to cache sets. However, existing schemes either require complex management logic, impose particular restrictions, rely on weak cryptographic functions, or mitigate only some classes of cache side-channel attacks. Most importantly, all of the aforementioned schemes are designed to provide side-channel cache protection for the entire code execution, which is actually not required in practice.

Our Goals. We observe that usually the majority of the code is not security-critical. Typically, a small portion of the code is security-critical and requires cache-based side-channel resilience. Moreover, this security-critical portion of the code is often already running in an isolated environment, such as in a TEE or in an isolated process. In these cases, a trusted component, namely the processor hardware or microcode or the operating system kernel, enforces this isolation. We aim to leverage and extend this existing isolation mechanism to also selectively enable side-channel resilience for the caches *only* for the portion of the code that needs it, without reducing the cache performance for the remaining non-isolated code. In doing so, we practically address the persistent performance-security trade-off of caches by providing the system administrator with a "tuning knob" to configure by balancing and isolating the workload as required. Consequently, s/he can tune the resulting cache side-channel resilience, utilization, and performance, while guaranteeing no performance overhead is incurred on the non-isolated portion of the code execution. Only the isolated (usually the minority) portion is subject to a reasonable reduction in cache capacity and performance – the cost of increased security guarantees.

To achieve this flexible and hybrid cache behavior, we introduce HYBCACHE, a generic mechanism that protects isolated code from cache side-channel attacks without reducing the cache performance for the remaining non-isolated code. In HYBCACHE, isolated execution only uses a pre-defined (small) number of cache ways[1] in each set of a set-associative cache. It uses these ways fully-associatively, while for eviction random victim cache lines are selected to be replaced by new ones, thus breaking the set-associativity and removing the root cause of access leakage. Non-isolated execution uses all cache ways set-associatively as usual, without any performance overhead. While isolated and non-isolated execution may compete for the use of some ways in the cache, the random replacement policy and fully-associative mapping used by the isolated execution prevent leaking information about the accessed memory locations (and their cache set mapping) to the non-isolated execution, thus making the precomputation and construction of an eviction set impossible. Moreover, HYBCACHE flexibly supports multiple, mutually distrusting isolated execution domains while preserving the above security guarantees individually for each domain.

HYBCACHE is architecture-agnostic, and can be seamlessly integrated with any isolation mechanism (TEEs or interprocess isolation); the definition of the isolation domains and the distribution of the workload is left up to the system administrator. HYBCACHE is backward compatible by design; it provides conventional set-associative caches for the workload if the side-channel resilience feature is not supported.

Contributions. The main contributions of this paper are as follows.

- We present HYBCACHE, the first cache architecture designed to provide flexible configuration of cache side-channel resilience by selectively enabling it for isolated execution without degrading the performance and available cache capacity of non-isolated execution.
- We evaluate the performance overhead of a simulator-based implementation of HYBCACHE and show that it is less than 5% for the SPEC2006 benchmarks suite,

[1] *Ways* are different available entries in a cache set to which a particular memory address can be allocated.

and estimate the memory and area overheads of a cycle-accurate hardware implementation of HYBCACHE.
- We show – through our security analysis – how breaking set-associative mapping and shared cache lines between mutually distrusting isolation domains (which are the root causes for typical cache side-channel attacks besides the intrinsic cache sharing and competition) mitigates typical contention-based and access-based cache attacks.

2 Cache Organization, Attacks and Defenses

We briefly present the typical cache organization, as well as recent cache side-channel attacks that are within the scope of our work, and limitations of existing defenses.

2.1 Cache Organization

Cache Structure. Caches are typically arranged in a hierarchy of fastest/closest/smallest to slowest/furthest/largest levels of cache, respectively L1, L2, and L3 cache/last-level-cache (LLC). Each core incorporates its L1 and L2 caches and shares the LLC with other on-chip cores. A cache consists of the storage of the actual cached data/instructions and the *tag* bits of their corresponding memory addresses. Cache memory is organized into fixed-size memory blocks, called *cache lines* each of size B bytes. Set-associative caches are organized into S sets of W ways each (called a W-way set-associative cache) where each way can be used to store a cache line. A single cache line can only be allocated to only one of the cache sets, but can occupy any of the ways within this cache set. The least significant $log_2\ B$ bits are the *block offset* bits that indicate which byte block within the B-Byte cache line is requested. The next $log_2\ S$ bits are the *index* bits used to locate the correct cache set. The remaining most significant bits are the *tag* bits for each cache line.

In a set-associative cache, once the cache set of a requested address is located, the *tag* bits of the address are matched against the tags of the cache lines in the set to identify if it is a cache hit. If no match is found, then it is a miss at this cache level, and the request is sent down to the next lower-level cache in the hierarchy until the requested cache line is found or fetched from main memory (cache miss). However, in a fully-associative cache, a cache line can be placed in any of the cache ways where the entire cache serves as one set. No index bits are required, but only $log_2\ B$ block offset bits and the rest of the bits serve as tag bits.

Eviction and Replacement. Due to set-associativity and limited cache capacity, cache contention and capacity misses occur where a cache line must be evicted in favor of the new cache line. Which cache line to evict depends on the replacement policy deployed, some of which include First-in-First-Out (FIFO), Least-Recently-Used (LRU), pseudo-LRU, Least-Frequently-Used (LFU), Not-Recently-Used (NRU), random and pseudo-random replacement policies. In practice, approximations to LRU (pseudo-LRU) and random replacement (pseudo-random) are usually deployed.

2.2 Cache Side-Channel Attacks

Cache side-channel attacks pose a critical threat to trusted computing and underlie more proliferating side-channel attacks such as the Spectre [44] and Meltdown [50] variants. Different classes of these attacks have been demonstrated on all platforms and architectures ranging from mobile and embedded devices [49] to server computing systems [34, 54, 81]. They have also been shown to undermine the isolation guarantees of trusted execution environments, like Intel SGX [11, 21, 60, 66] and ARM TrustZone [49, 80]. Such attacks have been shown to infer both fine-grained and coarse-grained private data and operations, such as bypassing address space layout randomization (ASLR) [23, 25], inferring keystroke behavior [26, 27], or leaking privacy-sensitive human genome indexing computation [11], as well as RSA [54, 81] and AES [10, 34] decryption keys.

Cache side-channel attacks exploit the inherent leakage resulting from the timing latency difference between cache hits and misses. This is then used to infer privacy/security-critical information about the victim's execution. In an offline phase, the attacker must first identify the target addresses of interest (by means of static and dynamic code analysis of the victim program) whose access patterns leak the desired information about the victim's execution, such as a private encryption key. In an online phase, the attacker measures the timing latency of its memory accesses or the victim's computation time to infer the desired information.

To demonstrate how a simple cache attack works, consider the pseudo-code of the Montgomery ladder implementation for the modular exponentiation algorithm shown in Algorithm 1. Modular exponentiation is the operation of raising a number b to the exponent e modulo m to compute $b^e\ mod\ m$ and is used in many encryption algorithms such as RSA. Leaking the exponent e may reveal the private key. As shown in Algorithm 1, the operations performed for each of the exponent bits directly correspond to the value of the bit. If the exponent bit is a zero, the instruction in Line 5 is executed. If the exponent bit is a one, the instruction in Line 9 is executed. An attacker that can observe or deduce these execution patterns can thus disclose the value of each corresponding exponent bit, and eventually recover the encryption key [78, 81]. S/he, however, needs to identify the target addresses that need to be observed (the addresses of the instructions in Lines 5 and 9 in this example) in the victim program and accordingly construct the eviction set. The eviction set is a collection of addresses that are mapped to the same specific cache set to which the target addresses are also mapped. The attacker uses this eviction set to evict the contents of the whole set in the cache, and therefore guarantee to successfully evict the target

addresses from the caches. Consequently, s/he measures the timing latency of its own memory accesses after a time interval to deduce whether the victim has accessed these target addresses.

Algorithm 1: Montgomery Ladder RSA Implementation

```
   Input: base b, modulo m, exponent e = (e_{n-1}...e_0)_2
   Output: b^e mod m
1  R_0 ← 1; R_1 ← b;
2  for i from n-1 downto 0 do
3      if e_i = 0 then
4          R_1 ← R_0 × R_1 mod m;
5          R_0 ← R_0 × R_0 mod m;
6      end
7      if e_i = 1 then
8          R_0 ← R_0 × R_1 mod m;
9          R_1 ← R_1 × R_1 mod m;
10     end
11 end
12 return R_0;
```

The online phase of these attacks consists of three main steps: *Eviction*, *Waiting* and *Analysis*. The attacker uses the eviction set to *evict* the victim's target addresses from the cache. Next, the attacker *waits* an interval of time to allow the victim to access the target addresses. Then the attacker measures and *analyzes* its access time measurements to determine if the victim has accessed the target addresses. This is repeated as many times as the attacker requires to collect sufficient traces to recover the exponent bits.

The different techniques used by the attacker to perform the eviction can be classified into two main approaches, either access-based or contention-based. In access-based attacks such as Flush + Reload [29, 78], Flush + Flush [26], Invalidate + Transfer [35], and Flush + Prefetch [25], the attacker accesses the target addresses directly by flushing them out of the cache using the dedicated *clflush* instruction [2] and possibly exploiting timing leakage from the execution of the *clflush* instruction [26]. This invalidates the lines containing these addresses and writes them back to memory. Evict + Reload [27] attacks have also been shown which do not require the *clflush* instruction, but instead evict specific cache sets by accessing physically congruent addresses. These attacks are only feasible in case of shared memory pages between the attacker and victim, usually in the form of shared libraries. Otherwise, an attacker resorts to contention-based attacks such as Prime + Probe [34, 38, 54, 61, 77], Prime + Abort [15], Evict + Time [23, 61], alias-driven attacks [28], and indirect Memory Management Unit (MMU)-based cache attacks [71], where s/he constructs an eviction set and uses it to trigger and exploit a cache contention in the same cache set as the target addresses, thus evicting cache lines containing the target addresses from the pertinent cache set.

The waiting interval should be selected and synchronized such that the victim is expected to access the target address at least once before the attacker analyzes the collected observations. By analyzing the collected observations, the attacker determines whether the target address was indeed accessed by the victim. This is achieved by different techniques depending on the attack approach, either the adversary measures the overall time needed by the victim process to perform certain computations [8, 10], or probes the cache with eviction sets and profiles cache activity to deduce which memory addresses were accessed [34, 38, 54, 77, 78], or accesses target memory addresses and measures the timing of these individual accesses [29, 61]. Alternatively, the adversary can also read values of addresses from the main memory to see whether cache lines that contain cacheable target addresses have been evicted to memory [28].

Cache-collision timing attacks exploit cache collisions that the victim experiences due to its cache utilization, e.g., after a sequence of lookups performed by a table-driven software implementation of an encryption scheme, such as AES [10]. These attacks are out of scope in this work since they are not common, are specific to certain software implementations, and can only be mitigated by adapting the implementation or locking the relevant cache lines after pre-loading them.

2.3 Limitations of Existing Defenses

To mitigate these attacks, software-based countermeasures and modified cache architectures have been proposed in recent years, which we cover in depth in the Related Work (Section 8). These can be classified into two main paradigms: 1) applying cache partitioning to provide strict isolation, or 2) applying randomization or noise to make the attacks computationally impractical. However, all proposed countermeasures to date either impact performance significantly, require explicit programmer's annotations, are not seamlessly compatible with existing software requirements such as the use of shared libraries, are architecture-specific, or do not defend against all classes of attacks. Most importantly, all existing defenses apply their side-channel cache protection for the entire execution workload.

In practice, cache side-channel resilience is only required for the security-critical (usually smaller) portion of the workload that is allocated to execute in isolation. Thus, non-isolated execution should not suffer any resulting performance costs. To address this in this work, we propose a modified hybrid cache microarchitecture that enables side-channel resilience only for the isolated portion of execution, while retaining the conventional cache behavior and performance for the non-isolated execution.

3 Adversary Model and Assumptions

To provide side-channel-resilient cache accesses for only security-critical isolated execution, we propose a hybrid *soft* partitioning scheme for set-associative memory structures.

In this work, we apply it to caches and call it HYBCACHE. HYBCACHE aims to provide cache-based side-channel resilience to the security-critical or privacy-sensitive workload that is allocated to one or more **I**solated Execution **D**omains (I-Domains), while maintaining conventional cache behavior for non-critical execution that is allocated to the **N**on-**I**solated Execution **D**omain (NI-Domain). HYBCACHE assumes an adversary capable of mounting the attacks described in Section 2.2 and is designed to mitigate them.

Furthermore, the construction of HYBCACHE is based on the following assumptions:

A1 Security-critical code that requires side-channel resilience is already allocated to an isolated component, like a process or a TEE (enclave).

A recent trend in the design of complex applications, like web browsers, is to compartmentalize them using multiple processes. As an example, all major browsers spawn a dedicated process for every tab [43] and some even use a dedicated process to better isolate privileged components [58]. Similarly, the widespread availability of TEEs, like SGX, encourages developers to encapsulate sensitive components of their code in protected environments.

A2 Isolated execution is the minority of the workload.

Isolation works best when the isolated component is as small as possible, thus reducing the attack surface. This complies with the intended usage of TEEs like SGX where only small sensitive components of the code would be allocated to the TEE. Hence, we assume only the minority of the workload needs to be isolated. HYBCACHE still provides the same security guarantees if the majority of the workload is isolated, but the performance of the isolated execution would suffer.

A3 Sensitive code only uses writable shared memory for I/O (if at all), and access patterns to this shared memory do not leak any information.

Isolated code should focus on processing some local data, while I/O needs should be limited to copying the input(s) into the isolated component, and copying the output(s) out of the component. Both of these procedures just access the data sequentially; thus, the access patterns during I/O do not depend on the data and does not leak any information.

A4 The attacker is not in the same I-Domain as the victim.

HYBCACHE is designed to isolate mutually distrusting I-Domains and thus, we must assume the attacker and the victim are not in the same I-Domain. Note that, as a consequence of A3, if a process handles sensitive data and has multiple threads, they must all be in the same I-Domain, since they share the entire address space. In cases where isolation between threads sharing the same address space is also required, HYBCACHE can, in principle, provide intra-process isolation as discussed later in Section 7.

4 Hybrid Cache (HYBCACHE)

We systematically analyzed existing contention-based and access-based cache attacks in the literature (Section 2.2) to identify their common root causes (besides the intrinsic sharing of cache entries and latency difference between a cache hit and miss). Cache side-channel attacks are, by nature, very specific to the victim program and may exploit attack-specific features such as the side-channel leakage of the *clflush* [26] or prefetch instructions [25]. Nevertheless, each one of these attacks is primarily caused by one or both of the following root causes: shared memory pages (and cache lines) between mutually distrusting code, and deterministic and fixed set-associativity of cache structures, which enables targeted cache set contention by pre-computed eviction sets.

4.1 Requirements Derivation

In light of the above, HYBCACHE should provide side-channel resilience between different isolation domains with respect to their cache utilization. An adversary process sharing the cache with a victim process should not be able to distinguish which memory locations a victim accesses. Nevertheless, we emphasize that the only approach to enforce complete non-interference between different domains is by strict static cache partitioning, such that no cache resources are shared, and thus zero information leakage occurs. On the other hand, this is impractical, and results in inefficient cache utilization from a performance standpoint. Our key objective in this work is to practically address and accommodate this persistent performance/security trade-off of cache structures by providing sufficiently strong cache side-channel-resilience, such that practical and typical cache side-channel attacks become effectively infeasible without necessarily enforcing complete non-interference. Additionally, we desire that this security guarantee is run-time configurable, such that it is only in effect when required.

This builds on our insight that it is neither practical nor required to provide cache side-channel resilience for all the code in the workload. This additional security guarantee is only required for security-critical execution, which is a minority of the workload (Assumption A2), and usually isolated in a Trusted Execution Environment (TEE) (Assumption A1). Thus, we require to provide a cache architecture that provides non-isolated execution with conventional cache utilization (with no performance costs), and simultaneously side-channel-resilient cache utilization (with a tolerable performance degradation) only for the smaller portion of the execution workload that is security-sensitive and isolated. We also require that our architecture is portable, can be easily deployed, and is backward compatible when a system does not support it. We summarize these requirements below:

R1 Strong side-channel resilience guarantees between the isolated and non-isolated execution domains, sufficient to

thwart typical contention-based and access-based cache attacks

R2 Dynamic and scalable cache isolation between multiple different isolation domains

R3 Addressing the cache performance/security trade-off by configuring the non-isolated/isolated workload balance (compliant with how TEEs are intended and designed to be used) such that the performance of the non-isolated execution workload is not degraded

R4 Usability: backward-compatible, architecture-agnostic, no usage restrictions and no code modifications required

Next, we present the high-level construction of HYBCACHE in Section 4.2 and its microarchitecture in more detail in Sections 4.3 and 4.4.

4.2 High-Level Idea

In HYBCACHE, a subset of the cache, named *subcache*, is reserved to form an orthogonal isolated cache structure. Specifically, $n_{isolated}$ *cache ways* within the conventional *cache sets* form the *subcache*. While these *subcache* ways are available for the NI-Domain to utilize, the I-Domains are restricted to utilize *only* these *subcache* ways. However, the I-Domains utilize this *subcache* in a fully-associative way and using a random-replacement policy. In doing so, all mutually distrusting processes executing in the I-Domains can share the *subcache* without leaking information on the actual memory locations they access. Since these *subcache* ways are not reserved exclusively for isolated execution and can also be utilized by non-isolated execution with least priority, the NI-Domain still retains unaltered cache capacity usage and non-degraded performance.

The key purpose of HYBCACHE, unlike existing defenses, is to selectively enable side-channel-resilient cache utilization only for the I-Domains. Hence, only the isolated execution is subjected to the resulting performance overhead, while still maintaining conventional cache behavior and performance for the NI-Domain, as outlined in Requirement R3. We describe next the architecture of HYBCACHE and how it achieves this.

4.3 Controller Algorithm

HYBCACHE modifies how memory lines are mapped to cache entries for the I-Domains. $n_{isolated}$ ways (at least a way in each set) of the conventional set-associative cache are designated to the orthogonal *subcache*. Cache lines are mapped fully-associatively to the *subcache* entries and evicted and replaced in the *subcache* using a random replacement policy. This means that a given memory line can be cached in any of the $n_{isolated}$ entries. This breaks the deterministic link between memory addresses and their corresponding cache locations, thus defeating an attacker that attempts to infer the victim's memory accesses by triggering and observing contention in a particular cache set.

Figure 1 illustrates how the HYBCACHE controller manages cache requests. HYBCACHE supports multi-core processors with simultaneous multithreading (SMT) and assumes that each process is assigned an `IsolationDomainID` (`IDID`) that identifies whether the process is in an I-Domain (and which isolation domain) or in the NI-Domain. Any incoming cache request is accompanied by the `IDID` of the issuing process. In Ⓐ, HYBCACHE controller queries the `IDID` of the cache request and the request is serviced accordingly. If it is in the NI-Domain, the complete cache is queried conventionally using the set index and tag bits of the requested address to locate the cache set and line respectively (Ⓑ & Ⓒ). If a match is found, the controller checks whether the cache line was found in one of the *subcache* ways in Ⓓ. Recall that these ways are not reserved exclusively for isolated execution, i.e., they can be used by non-isolated execution but with least priority in case a cache set becomes over-utilized. Therefore, if a matching cache line is found in one of these ways, the controller checks whether it was cached by an isolated or non-isolated process (Ⓔ). The requesting process can only hit and access the cache line if that line was placed by a process in the NI-Domain. Otherwise, it is not allowed to hit on it.

Checks in the controller are implemented to occur in parallel, i.e., all cache hits are generated in the same number of clock cycles (as well as cache misses), to eliminate respective timing side channels. In case of a cache miss, the memory block is fetched from main memory and cached in Ⓕ. The eviction and replacement are performed according to the deployed policy. All ways are available for eviction, including the *subcache* ways to provide the NI-Domain execution with unaltered cache capacity. However, the usage of the *subcache* ways by the I-Domains is considered while recording the recency of accesses to the cache ways to make it least likely to evict a line from one of the *subcache* ways if it is recently used by an I-Domain process.

If the cache request is issued by an I-Domain process, it is serviced by querying only the *subcache* (Ⓖ). The *subcache* deploys fully-associative mapping, and is thus queried by a lookup of all the ways using the (cache line address bits - block offset bits) as tag bits (Ⓗ) and simultaneously querying that the line belongs to an I-Domain (since these ways may also be used by the NI-Domain) and that it was placed by a process with the same `IDID` (Ⓘ). Otherwise, a cache miss occurs. Disallowing I-Domain processes from hitting on cache lines originally placed by processes in other I-Domains provides dynamic isolation between an unlimited number of mutually distrusting processes that share memory. In case of a miss, any of the *subcache* ways is randomly selected and its cache line is evicted and replaced by the memory block fetched from main memory (Ⓙ). The random replacement policy considers all *subcache* ways equally, even those occupied by the NI-Domain cache lines.

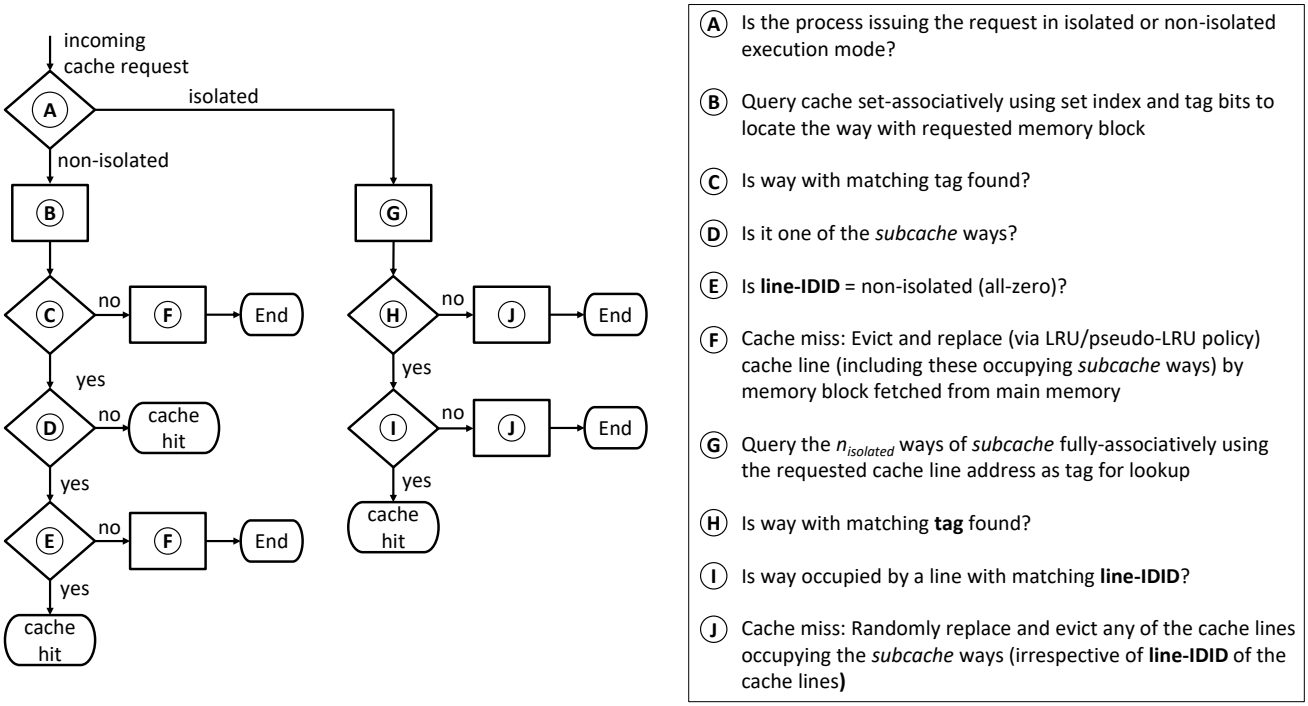

FIGURE 1: HYBCACHE controller policy

4.4 Hardware Microarchitecture

Figure 2 shows how HYBCACHE could be applied for a conventional cache hierarchy of a multi-core processor. The cache capacity available for the NI-Domain execution is unaltered, i.e., the conventional set-associative cache with all its sets and ways can be utilized by the NI-Domain.

At each cache level, way-based partitioning is used to reserve at least a way in each set (gray ways in Figure 2). These ways, combined, form the orthogonal *subcache* that the I-Domain execution is restricted to use. However, these *subcache* ways are *not* used exclusively by the I-Domain execution, i.e., the NI-Domain execution may use these ways in case a corresponding set is fully utilized and the least-recently-used (LRU) replacement algorithm requires to evict a cache line from a *subcache* way in this set. This ensures that the NI-Domain execution is provided with unaltered cache capacity and does not suffer performance degradation.

The *subcache* is fully-associative and deploys random replacement policy, i.e., a given memory block is always equally likely to be cached in any of the available ways. This breaks set-associativity and provides randomization-based dynamic isolation between different I-Domains while allowing flexible sharing of the *subcache* depending on the run-time utilization requirements of the isolated execution domains. Using the *subcache* fully-associatively further maximizes the utilization of its limited hardwired capacity.

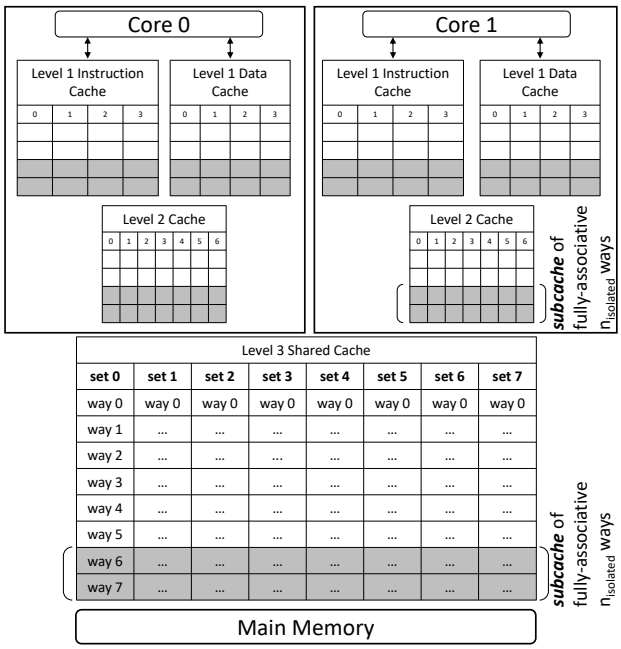

FIGURE 2: HYBCACHE hierarchy and organization

The $n_{isolated}$ ways that form the *subcache* are configured (hardwired) at design-time and cannot change at run-time, because these ways are members of both the primary cache as

well as the *subcache* as shown in Figure 3. It is not feasible to make $n_{isolated}$ run-time configurable, as this would require that *all* the ways are unreasonably wired in both a fully-associative and set-associative organization. Thus, only a small subset of $n_{isolated}$ ways (dark gray ways in Figure 3) is selected to form the *subcache*. Each of the *subcache* ways is augmented with `IsolationDomainID` (`IDID`) configuration bits to identify the isolation domain that placed an occupying cache line in the pertinent way. To provide any cache isolation at the microarchitectural level, a mechanism to bind owners/tags to cache lines is required, thus `IDIDs` are needed. We chose to configure 4 bits for the `IDID`, thus supporting 16 concurrent isolation domains, where an all-zero indicates the NI-Domain. The number of bits allocated in HYBCACHE for `IDID` is a hardware design decision. Increasing the number of designated bits would increase the number of maximum concurrent isolation domains that HYBCACHE can support. However, other metrics such as area overhead and power consumption come into play in this design trade-off.

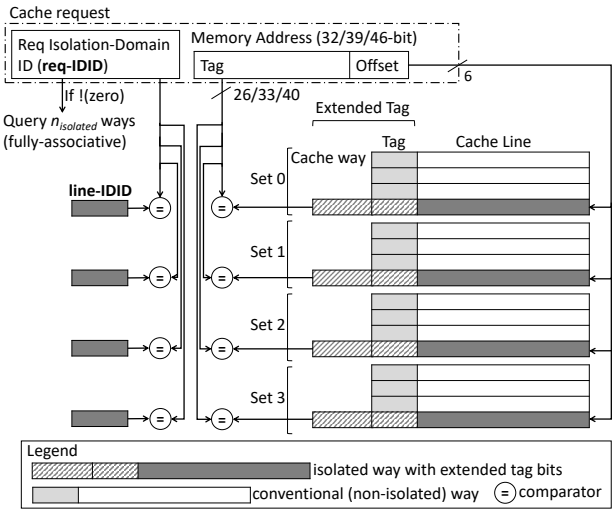

FIGURE 3: HYBCACHE hardware microarchitecture

The *subcache* ways are augmented with an extended tag bits storage (dashed dark gray tag bits of the dark gray ways in Figure 3). When queried fully-associatively (for the I-Domains), all bits, except the offset bits (6 bits for byte-addressable 64B cache line), of the requested address are compared with the extended tag bits of the *subcache* ways to locate a matching cache line. For the NI-Domain, the *subcache* ways are queried set-associatively with the rest of the cache (conventionally), where the request tag bits are compared only with the non-extended tag bits of the *subcache* ways within the located cache set.

4.5 Software Configuration

Abstraction and Transparency. The hardware modifications required for HYBCACHE are transparent to the software and abstracted from it. The trusted software (or hardware) component of the incorporating platform is only required to interface with the HYBCACHE controller to communicate the isolation domain of each incoming cache request. However, HYBCACHE does not stipulate or restrict how these isolation domains are defined and communicated, thus leaving it to the discretion of the system designer to identify how HYBCACHE can be integrated with the comprising architecture.

Isolated Execution. HYBCACHE enables the dynamic isolation of the cache utilization of different isolation domains by using the `IDID` of the process that issues the cache request being serviced. The means by which the isolation domains are defined, generated, and communicated is dependent on how the trusted execution and isolation is deployed. We design HYBCACHE such that it is seamlessly compliant with any trusted execution environment (TEE) where isolation domains (across different processes, cores, containers, or virtual machines (VMs)) are either software-defined by a trusted OS (thus requiring kernel support) or hardware/firmware-defined in case the OS is not trusted (such as in SGX). Different isolation domains can be defined across different isolated address space ranges such as in SGX enclaves, across processes such as in TrustZone normal/secure worlds or by standard inter-process isolation, or even across different groups of processes or different virtual machines.

HYBCACHE is agnostic to the means of defining the `IDIDs` of different isolation domains, and complements any form of isolated execution environment in place to provide it with cache side-channel resilience. If the kernel is trusted, kernel support is required to assign an `IDID` (or an all-zero `IDID` for a non-isolated process) to each process according to its isolation domain. The `IDID` bits can be added as an additional process attribute in each process's process control block (PCB). Otherwise, the trusted hardware or firmware would assign the isolation domains. HYBCACHE assumes that some mechanism of isolation is already enforced for security-critical code that it can leverage to provide the cache-level isolation. We argue why this is reasonable in Assumption A1. Nevertheless, if this is not the case, then isolation domains need to be explicitly defined by the developer if s/he wishes to protect particular code against cache-based side-channel attacks. While HYBCACHE is focused on protecting user code, in principle, kernel code can also be protected by allocating it to an isolation domain.

Backward Compatibility. Similar to processor supplementary capabilities such as Page Attribute Tables (PATs) and Memory Type Range Register (MTRR) for x86, HYBCACHE supports providing side-channel-resilience on-demand while

retaining backward compatibility. HYBCACHE only effectively provides side-channel resilience for the cache utilization of execution when processes are assigned different IDIDs that are communicated with each cache request. Otherwise, from a software perspective, HYBCACHE is identical to a conventional cache architecture. If no isolation domains are assigned to the different processes by the trusted kernel or trusted hardware, HYBCACHE is designed to assign an all-zero IDID by default to incoming cache requests and all execution is treated as non-isolated (see Figure 1) with cache-based side-channel resilience disabled. Only when kernel support is provided (or trusted hardware or firmware in case of SGX) does HYBCACHE behave differently for different isolation domains and provides its side-channel resilience capability.

Shared Memory Support. HYBCACHE supports, by design, that different isolation domains can share read-only memory, usually in the form of shared code libraries, without sharing the corresponding cache lines. This results in having multiple copies of the shared memory kept in cache (multiple cache entries), enforcing that cache entries are not shared between mutually distrusting code. Data coherence is also not a problem, in this case, since this is read-only memory. We elaborate in Section 5 how this effectively mitigates access-based side-channel attacks.

Conventional access to shared writable memory, on the other hand, between different isolation domains is disallowed by design in HYBCACHE, as this makes the victim process vulnerable to access-based attacks and would undermine cache coherence. In order to provide input and output functionality to isolated code, HYBCACHE provides special *I/O move instructions*. These allow code in an I-Domain to transfer data between a CPU register and a memory region (assigned an all-zero IDID when cached) that is designated exclusively for shared memory between processes belonging to different I-Domains. These special instructions are meant to be used to transfer data between domains only through this designated memory. In practice, we expect them to be used only in frameworks like the SGX SDK or a trusted kernel. If code in an I-Domain incorrectly accesses this memory region using regular instructions, or accesses its own memory using these special instructions, this could be disallowed, i.e., detected and blocked by the hardware or microcode, e.g., the MMU. This prevents inserting duplicated writable cache entries which can disrupt cache coherency, while ensuring that HYBCACHE's security guarantees still apply to any access performed using regular instructions.

5 Security Analysis

In the following, we evaluate the effectiveness of HYBCACHE with respect to the security requirements we outlined in Section 4.1. We show that HYBCACHE achieves these security guarantees by mitigating the following leakages:

S1 Malicious software running in an I-Domain or NI-Domain cannot flush or perform a cache hit on a cache line belonging to a different I-Domain.

S2 Malicious software running in an I-Domain or NI-Domain cannot pre-compute and construct an eviction set that selectively evicts a non-trivial subset of the cache lines belonging to a different I-Domain. Moreover, the set of the attacker's cache lines which can be evicted by the victim's lines does not depend on the addresses accessed by the victim.

S3 Cache hits generated by software in an I-Domain cannot be observed by software running in a different I-Domain or NI-Domain. Cache misses generated by software in an I-Domain can still be indirectly observed by malicious software running in a different I-Domain or NI-Domain, but the malicious software learns no information (e.g., memory address) about the access besides whether a cache miss has occurred.

5.1 S1: Absence of Direct Access to Cache Lines

Access-based attacks, like Flush + Reload [29, 78], Flush + Flush [26], Invalidate + Transfer [35], Flush + Prefetch [25], and Evict + Reload [27], require the attacker to have direct access to the victim's cache lines, normally as a result of shared memory between processes (e.g., shared libraries). As an example, Flush + Reload works by flushing shared cache lines and monitoring which lines the victim accesses and brings back into the cache. HYBCACHE mitigates this class of attacks by preventing shared cache lines between the attacker and victim, as we explain in the following.

Shared Read-Only Memory. Read-only memory is shared between different processes in case of shared code libraries. HYBCACHE provides support for shared read-only memory (Section 4.5), while fundamentally disallowing that any cache line is shared across different I-Domains. Execution within one domain can only access cache lines brought into the cache by the same domain. Separate (potentially duplicate) cache lines are maintained for each domain; flushing and reloading cache lines only impacts those owned by the attacker's domain and cannot influence any other I-Domain or leak any information on its cache lines. Having duplicate cache lines for read-only memory pages does not disturb cache coherency because it is read-only.

Shared Writable Memory. Shared writable memory between mutually distrusting domains is disallowed by design with HYBCACHE. Code in an I-Domain can only exchange data with another isolation domain through the special I/O

move instructions, which transfer data between the CPU registers and memory in the NI-Domain that is designated for shared communication (see Section 4.5). Incorrect usage of those instructions or incorrect access to this designated memory region could be detected and blocked by the MMU to prevent potential cache coherency disruption due to duplicate writable cache entries. However, HYBCACHE still enforces that every cache line only belongs to one domain. Since cache lines always belong to one specific I-Domain or the NI-Domain, code in a domain cannot flush or perform a cache hit on a different domain's cache lines (S1), and attacks that rely on those capabilities are thus impossible.

5.2 S2: Impossibility of Pre-Computed Eviction Set Construction

Without direct access to the victim's cache lines, attackers resort to contention-based attacks, like Prime + Probe [34, 38, 54, 61, 77], Prime + Abort [15], and Evict + Time [23, 61]. In these attacks, the attacker pre-computes and constructs an eviction set which ensures eviction of a specific subset of the victim's cache lines, e.g., lines that belong to a specific set in a set-associative cache. The attacker process first accesses the whole eviction set, thus ensuring the victim's cache lines are evicted. After a waiting interval, it then checks if its whole eviction set is still in cache by timing its own memory accesses to this set, thus detecting if the victim accessed any of the cache lines of interest. For a conventional set-associative cache, this is possible because of a fixed set-indexing, which can be directly determined from the target address of interest.

HYBCACHE protects I-Domains from such attacks by disabling the set-associativity of the reserved *subcache* entries when they are used by isolated execution: when a memory address is accessed by the isolated victim process, the cache line will be stored in any entry chosen randomly from the whole *subcache* and not from a specific set. The random replacement policy for isolated execution ensures that any of the *subcache* entries is chosen using a discrete uniform distribution, i.e., with an equal and independent probability every time, so the attacker has no means of identifying deterministically and reproducibly which cache set (or entry) will be used to cache a particular memory access of the victim. In order to ensure that a specific cache line of the victim is evicted, the attacker can only evict all lines in the *subcache*, but s/he cannot selectively evict a non-trivial subset of the victim's cache lines. Moreover, the set of the attacker's cache lines which can be evicted by the victim's lines does not depend on the addresses accessed by the victim (S2). As a consequence, attacks that rely on these capabilities are no longer possible. This holds whether the attacker process is running in an I-Domain or NI-Domain, as long as the victim process is in an I-Domain (Requirements R1 and R2).

5.3 S3: Observable Cache Events

Software running in an I-Domain can only hit on cache lines belonging to the same I-Domain. These cache hits generate no changes to the cache state, thus, they are unobservable by an attacker in a different I-Domain or in the NI-Domain.

Cache misses generated by software in an I-Domain evict a random cache line, which may belong to a different I-Domain or the NI-Domain. Malicious attacker code can then periodically observe how many of its lines are evicted and infer the number of cache misses the victim process is experiencing. The attacker can further use this information to infer the size of the victim's working set, i.e., the number of cache lines in the *subcache* currently belonging to the victim.

This cache occupancy channel is the only side-channel leakage that is not mitigated by the HYBCACHE construction, which is inherently available in any cache architecture where the attacker and the victim processes compete for entries in shared cache resources. It can only be effectively blocked by strict cache partitioning, which we deliberately do not provide in the HYBCACHE construction. This allows different isolation domains to still compete for cache entries, thus preserving maximum and dynamic cache utilization and unaffected performance for non-isolated execution, as our performance evaluation shows in Section 6.1. Note that, due to S2, the information inferred by the attacker from observing this remaining leakage, is effectively reduced to only knowing the working set size at any point in time.

Leveraging this side channel to infer further information and mount an attack in typical settings is not trivial. The victim may evict its own lines when it experiences cache misses due to the random replacement policy. This would not effect a difference in the cache state for the attacker, which complicates the attacker's bookkeeping. Moreover, observations are severely hindered when any other software is concurrently running besides the attacker and the victim processes. Finally, standard software hardening techniques can be applied to mitigate attacks to code implementations that are particularly sensitive to this attack. Furthermore, exploiting this side channel to leak data has not been shown in practice. A recent attack [67] leverages the cache occupancy side channel to infer which website is open in a different browser tab (under the strong assumption that no other tabs are open); however, it does not leak any user data. Cache activity masking is suggested as one of the countermeasures to the attack. Implementing cache activity masking for HYBCACHE is feasible and independent of our cache architecture.

Since the attacker aims to maximize its information and cannot observe cache hits, s/he can attempt to evict all *subcache* entries in order to maximize the number of misses experienced by the victim. As we discuss later, evicting the whole *subcache* takes time for an attacker in either the NI-Domain or in a I-Domain. An unprivileged attacker is unable to pause the victim's execution; thus, the attacker can only measure the

cache usage with limited granularity. However, a privileged adversary, like a malicious OS in the case of an SGX enclave, can stop and restart the victim arbitrarily and leverage tools like SGX-Step [12] to observe the victim's cache usage with fine granularity. HYBCACHE does not mitigate such an attack by construction. However, mitigating it is only possible by strict cache partitioning and the resulting performance costs. We emphasize that we make an intentional design decision in HYBCACHE to allow isolation domains to dynamically compete for cache entries for maximum cache utilization and unaffected performance for non-isolated execution. A HYBCACHE construction that dynamically allocates a dedicated *subcache* for each isolation domain would block this leakage and mitigate attacks that rely on it.

Non-isolated Attacker Process. If the attacker process is in the NI-Domain, in order to guarantee eviction of the whole *subcache* it must fill up all ways in every cache set, including the *subcache* ways. Therefore, the attacker process must construct an eviction set that is as large as the entire cache capacity. A typical data L1 cache holds 512 cache entries. In our experiments, probing (accessing and measuring access latencies) of 512 cache lines takes approximately 30 000 CPU cycles, i.e., a little over 8 µs.[2] For larger caches, such as the LLC, it is not even feasible to mount Prime+Probe attacks by probing the entire cache. The adversary is required to pinpoint a few cache sets that correspond to the relevant security-critical accesses made by the victim and monitor these only [54].

Isolated Attacker Process. If the adversary is in a different I-Domain than the victim process, it still cannot control cache eviction of particular target addresses specifically. Both attacker and victim processes are isolated and can only use the *subcache* ways. Thus, an adversary aiming to perform controlled eviction can only try to evict the entire *subcache*. Because the *subcache* is fully-associative with random replacement, evicting the entire *subcache* requires an eviction set much larger than the *subcache* capacity. We argue below that this is not easier than probing the entire L1 cache (in case the attacker is non-isolated), for instance, even though the *subcache* is significantly smaller. Moreover, it can be only guaranteed up to a certain level of probabilistic confidence. This can be represented statistically by the coupon collector's problem, where coupons are represented by entries in the *subcache*. Let $N_{accesses}$ be the total number of accesses needed to evict all the *subcache* entries n and n_i be the number of accesses needed to evict the i-th way after i-1 ways have been evicted. Both $N_{accesses}$ and n_i are discrete random variables. The probability of evicting a new way becomes $\frac{(n-(i-1))}{n}$. The expected value and variance of $N_{accesses}$ are

$$\mathbb{E}(N_{accesses}) = n \cdot \mathrm{H}_n \qquad \mathbb{V}(N_{accesses}) \approx \frac{\pi^2}{6} \cdot n^2$$

H_n denotes the n^{th} harmonic number. For $n = 128$ *subcache* entries, an average of 695 memory accesses (each mapping to a different 64B cache line) is needed to evict the *subcache* with a variance of $\approx 26\,951$. This is comparably more than the 512 accesses required to probe the entire typical L1 cache if the attacker process is not isolated (see above). Moreover, with such a large variance, significant variations in the number of $N_{accesses}$ required are expected from the mean $\mathbb{E}(N_{accesses})$ every time this eviction process is repeated.

6 Evaluation

Cache	Size	Associativity	Sets
L1	64 KB	8-way associative	128
L2	256 KB	8-way associative	512
L3	4 MB	16-way associative	4096

TABLE 1: Cache hierarchy used in our evaluation

Mix	Components
`pov+mcf`	povray, mcf
`lib+sje`	libquantum, sjeng
`gob+mcf`	gobmk, mcf
`ast+pov`	astar, povray
`h26+gob`	h264ref, gobmk
`bzi+sje`	bzip2, sjeng
`h26+per`	h264ref, perlbench
`cal+gob`	calculix, gobmk
`pov+mcf+h26+gob`	povray, mcf, h264ref, gobmk
`lib+sje+gob+mcf`	libquantum, sjeng, gobmk, mcf

TABLE 2: Benchmark mixes used in our evaluation

HYBCACHE is architecture-agnostic and applicable to x86, ARM or RISC-V. We performed our performance evaluation of HYBCACHE on a gem5-based [9] x86 emulator. We evaluated the hardware overhead for an RTL implementation that we implemented to extend an open-source RISC-V processor Ariane [62]. For our prototyping, we applied HYBCACHE to L1, L2, and LLC. We describe our evaluation results next.

6.1 Performance Evaluation

To evaluate HYBCACHE, we chose eight mixes of programs from the SPEC CPU2006 benchmark suite, which are used in the literature[3] [36, 76], shown in the upper part of Table 2.

[2] We ran this experiment on an Intel i7-4790 CPU clocked at 3.60 GHz.

[3] [76] also uses a ninth mix, `dea+pov`, which fails to run on gem5.

Two-Process Mixes. In order to evaluate the impact of isolating one process in the context of an SMT processor, we configure gem5 to simulate two processors connected to a single three-level cache hierarchy, whose parameters are shown in Table 1. The caches have the latencies used in [76].

For each mix, we first isolate one process, then the other, and we compare the performance of those processes to a third run in which neither process is isolated. We make either 2 or 3 of ways per set usable by the isolated execution processes. The replacement policy for non-isolated processes is LRU. Like in [76], we let gem5 simulate the first 10 billion instructions of each process in order to let the process initialize, then we measure the performance of one additional billion instructions. We measure the performance overhead as the relative change in the instructions-per-cycle (IPC), i.e., the ratio between instructions executed and CPU cycles required. A *positive* overhead represents a *decrease* in performance.

Figure 4 reports the IPC overhead of each program when running in isolation mode, while the other member of the mix runs in normal mode, for 2 or 3 isolated ways. The geometric mean of the positive overheads is 4.95% with 2 isolated ways and 3.47% with 3 isolated ways, with maximum overheads of 16% and 14% respectively for the `cal+gob` mix. For this mix, the overhead is due to a significantly increased L3 cache miss rate: the data miss rate jumps from 0.6% to 17.6%, while the instruction miss rate increases from 2.1% to 9.0%. The working set of `calculix` normally fits in L3 [36] but it does not in the *subcache*, hence the higher overhead. Since HYBCACHE is meant to protect only sensitive applications, which can be expected to be short-lived and only constitute a minority of the workload of a system, we consider those overheads easily tolerable. Figure 5 reports the IPC overhead for the member of the mix that is not isolated. In all cases the IPC overhead is not positive, i.e., the IPC is equal or better than the baseline, thus showing that HYBCACHE does not degrade the performance of non-isolated processes.

Four-Process Mixes. To demonstrate scalability, we also ran four-process mixes, shown in the bottom part of Table 2. We configured gem5 with four cores; two cores share an L1 and L2 cache, the other two cores share one additional L1 and L2, while L3 is shared by all cores. Isolated execution can use two ways per set. We isolated each member of the two mixes (the first eight bars in Figure 6), while the other three processes were running normally. Each isolated process has an overhead similar to that reported in the two-process mix experiments in Figure 4. Moreover, we also isolated two processes in each mix (last two columns in Figure 6). In this case, we measured increased overheads by up to 2 additional percentage points due to the additional competition for the *subcache*. However, those overheads are still easily tolerable given the security benefits and that they are only incurred by the isolated execution.

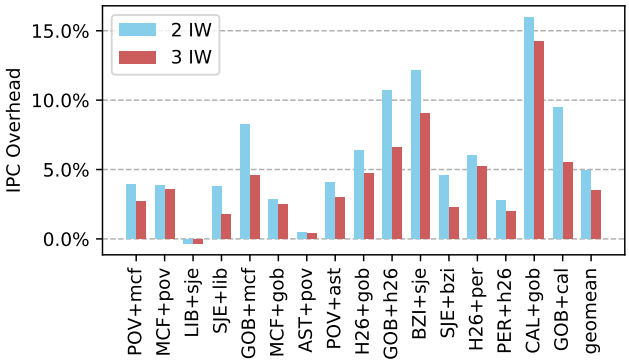

FIGURE 4: IPC overhead of each isolated process when 2 or 3 ways are available to isolated execution. Each pair of bars refers to a specific 2-process mix: the uppercase benchmark is isolated and the other is not.

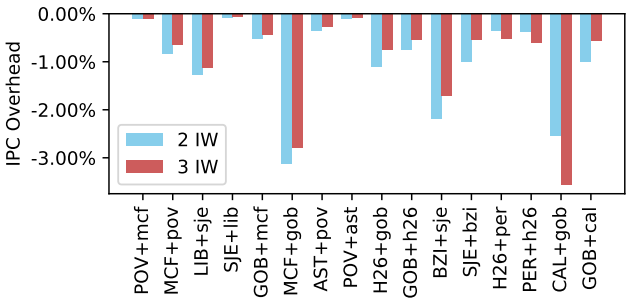

FIGURE 5: IPC overhead of each process when the other member of the mix is isolated. Each pair of bars refers to a specific 2-process mix: the uppercase benchmark is isolated and the other is not.

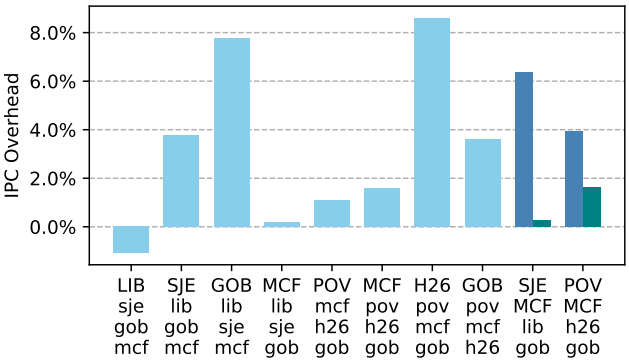

FIGURE 6: IPC overhead of isolated processes for 4-process mixes. The uppercase benchmarks are isolated and the others are not. The last two columns have two bars each since two process are isolated.

$n_{isolated}$	NAND2X1 Gates	Memory Overhead (Kb)
32	6114	0.34
64	12219	0.68
128	24563	1.3
256	48796	2.75
512	97830	5.5
1024	201792	11
2048	458300	22

TABLE 3: Logic and memory overhead estimates for fully-associative lookup of 46-bit addresses for different numbers of isolated cache ways (in any cache level).

6.2 Hardware and Memory Overhead

HYBCACHE requires additional hardware and memory for the fully-associative lookup of the *subcache* entries. We implemented the RTL for HYBCACHE and evaluated it for the hardware overhead for different number of isolated cache ways as shown in Table 3, irrespective of which cache levels this is applied to. While the overhead of the additional hardware is non-negligible, it is reasonable for a fully-associative cache lookup. Nevertheless, it diminishes in perspective with an 8-core Xeon Nehalem [1] of 2,300,000,000 transistors, for example. The logic overhead of HYBCACHE for 2048 fully-associative ways lookup is estimated at 1,833,200 transistors (NAND2X1 count $\times$ 4) which is 0.07% overhead to the Xeon Nehalem. For an 8-way 128-set cache, the memory overhead in our PoC for fully-associative mapping is 7 additional tag bits + 4 `IDID` bits per cache way. With respect to access latencies, the exact timing latency of lookups will eventually depend on the circuit routing but, in principle, for a parallel content-addressable memory lookup (as in our hardware PoC), accesses are performed in 2 clock cycles.

7 Discussion

Design and Implementation Aspects. HYBCACHE relies on a random-replacement cache policy combined with full-associativity to provide its dynamic isolation guarantees. The implementation of the random replacement policy is delegated to the hardware designer and considered an orthogonal problem. Cryptographically-secure pseudo-random number generators (CSPRNG) or even true hardware random number generators can be used and the seed can be changed as often as required. The output of the CSPRNG cannot be predicted if it is seeded with secret randomness at the start of every process. When the seed is changed, re-keying management tasks such as cache flushing and invalidation for the re-mapping are not required, unlike in recent architectures [63, 74]. This is because in HYBCACHE the randomness is only used for selection of the victim cache line, and not for locating existing cache lines in the *subcache*. Furthermore, we emphasize that CSPRNG design and implementations are an orthogonal problem to our work.

The "soft" cache partitioning of HYBCACHE is a generic concept and can be applied, in principle, to any set-associative structure. In this work, we apply it to the L1, L2, and L3 (LLC) caches, but it can also be applied selectively to only some of these cache levels or to the TLB as well, or to only some cache levels in only one or more cores in a multi-core architecture that become dedicated for allocating isolated execution. The choice of which cache structures to apply this to and how many ways to isolate in the *subcache* is delegated to the hardware designer, given that it is a more complex design decision with other metrics and trade-offs that come into play such as the size of the structure, power consumption, and logic overhead. The power consumption and timing overheads associated with building and routing a fully-associative cache lookup in VLSI are significant, but can be alleviated by leveraging emerging hybrid memory technologies such as DRAM-based caches [48] and STT-MRAM caches [30, 31]. In practice, applying HYBCACHE to the LLC or larger caches in general would be more expensive (in terms of hardware) than L1 and L2 caches, and strict partitioning might be applied instead for the LLC. Nevertheless, HYBCACHE can be, in principle, applied to sliced Intel LLCs. In each slice, a number of cache ways (*subcache*) is reserved for isolated execution. Any mapping from the `IDID` to the LLC slices can be used, such that lines from a particular `IDID` are allocated to a specific slice. Fully-associative lookups are thus only be performed on the *subcache* portion of a single slice, thus reducing the performance overheads and allowing scaling to high-core-count processors. The slice-mapping would be based only on the `IDID`, and thus it would not leak any information about the data address or value.

Other design decisions in HYBCACHE include the number of bits designated for `IDID` and thus the maximum number of concurrent isolation domains supported (see Section 4.4). To support more isolation domains (not concurrently) than the hardwired maximum, the cache lines of one domain can be flushed by the kernel or microcode at context switching while the next domain is switched in and is re-assigned the available `IDID`. Nevertheless, supporting too many isolation domains will result in increased cache utilization, and the overall performance will suffer. This is in line with conventional cache behavior, but is aggravated in HYBCACHE because isolated execution is only allowed to utilize the *subcache* portion. However, this violates our working assumption A2 that only the minority of the workload requires cache-level isolation.

We emphasize that cache-based side-channel leakage directly results from the design of the cache microarchitecture and, thus, it is reasonable to investigate the fundamental microarchitectural designs of caches for upcoming processor designs. While this does not address the problem for legacy systems, it provides an exploratory ground of ideas for upcoming processor designs. HYBCACHE is architecture-agnostic

and can be integrated with any processor architecture (we simulated it for x86 and implemented it for RISC-V). It is also compliant with any set-associative cache architecture independent of its hierarchy and organization, and whether it is virtually or physically indexed since no indexing is involved.

Intra-Process Isolation Support. HYBCACHE can also be extended, in principle, to provide *fine-grained* run-time configuration of the isolation domain *within* a process, e.g., between different threads within the same process. Besides kernel support, this requires an instruction extension to enable isolation of particular code regions or threads to different `IDID`s or disable isolation altogether at run-time (reset its run-time `IDID` to all-zero). However, this requires the developer to identify and annotate security-sensitive code regions. Nevertheless, this is useful in practice since a process might not require cache-based side-channel resilience for its entirety but only for sensitive code such as cryptographic computations. This is a more generalizable approach that is easier and more directly applicable than implementing leakage-resilient variants for security/privacy-sensitive computations.

Deployment Assumptions. HYBCACHE assumes any TEE or trusted computing environment that is leveraged in compliance with their original design intent, i.e., that the much larger portion of the execution workload is not security-critical and only a smaller portion is security-critical and isolated in an I-Domain (A2). Otherwise, if the workload is equally balanced, the isolated execution subset would be restricted to a smaller partition of the cache and would incur a more than tolerable performance degradation especially if it is cache-sensitive. For HYBCACHE to be optimally advantageous, the workload distribution and allocation must be performed by the administrator such that the right balance of overall security and performance is achieved, as shown by the performance results in Section 6.1.

8 Related Work

We describe next the state of the art in existing defenses and their shortcomings that HYBCACHE overcomes.

8.1 Partitioning

Cache partitioning allocates to each process or security domain a separate partition of the cache, hence guaranteeing strict non-interference. Both software-based [20, 40, 51, 82] and hardware-based [24, 41, 72, 73] partitioning schemes have been proposed in recent years, where partitioning is either process-based or region-based.

Process-based partitioning. Godfrey [20] implements process-based cache partitioning using page coloring on Xen, which incurs a prohibitive performance overhead with increasing number of processes. SecDCP [72] is a way-partitioning scheme where each application is assigned a security class and cache partitioning between the security classes is dynamically managed according to the cache demand of non-secure applications. SecDCP is not scalable; selective cache flushing and repartitioning is required if the number of security classes exceeds that of allocated partitions and it may perform worse than static partitioning. Furthermore, both schemes do not support the use of shared libraries. CacheBar [82] periodically configures the maximum number of ways allocated to each process which unfairly impacts performance and cache utilization, and does not scale well with the number of security domains. DAWG [41] partitions the caches where different processes are assigned to different protection domains isolating cache hits and misses. The aforementioned schemes incur the performance overhead for the entire code, whereas HYBCACHE only enables side-channel resilience and the resulting performance overhead only for the isolated execution.

Sanctum [14] protects TEEs by flushing private caches whenever the processor switches between enclave mode and normal mode and partitioning of the LLC and assigning to each enclave a static number of sets. Sets allocated to an enclave can be used exclusively by the enclave and cannot be utilized by the OS. On the contrary, HYBCACHE allows for a flexible and dynamic sharing of cache resources between processes (thus improving performance), while preserving cache side-channel resilience for isolated execution.

Many cache partitioning and allocation schemes [37, 55, 64, 65, 75] have been proposed that focus on cache allocation mechanisms aiming to improve performance for multi-core caches. However, such schemes do not provide security guarantees. HYBCACHE addresses the security/performance trade-off by providing a configurable means to enable the side-channel resilience only for isolated execution while providing non-isolated execution with unaltered performance.

Region-based partitioning. These approaches split the cache into a secure partition reserved for security/privacy-critical memory pages and a non-secure partition for the remaining memory pages. STEALTHMEM [40] uses page coloring where several pages are colored and reserved for security-sensitive data and they remain locked in cache. CATalyst [51] leverages Intel's CAT (Cache Allocation Technology) [3] to divide the cache into secure and non-secure partitions and uses page coloring within the secure partition to isolate different processes' cache accesses to these pages. PLcache [73] locks cache lines and allocates them exclusively to particular processes such that the cache line can only be evicted by its process. However, overall performance and fairness of cache utilization are strongly impacted as the protected memory size increases in relevance to the total cache capacity. Moreover, with PLcache an attacker process may still infer the victim's memory accesses by observing that it is unable to access or evict cache lines (locked by a victim process) from a particular cache set.

Cloak [24] uses hardware transactional memory, such as Intel TSX [2], to protect sensitive computations by pre-loading the security-critical code and data into the cache at the begin-

ning of the transaction and any cache line evictions are detected by the transaction aborting. Cloak incurs prohibitively high performance overhead for memory-intense computations and requires the developer's strong involvement to identify and instrument security-sensitive code and split it into several transactions. Recent works have also explored the LLC inclusion property for defense schemes such as RIC [39] and SHARP [76]. However, both are architecture-specific, RIC requires coherence protocol modifications and cache flushing on thread migration, while SHARP requires modifications to the *clflush* instruction. HYBCACHE, however, is architecture-agnostic, and does not require cache flushing or modifications to coherence protocols or the *clflush* instruction.

8.2 Randomization

Introducing randomization involves introducing noise or deliberate slowdown to the system clock to hinder the accuracy of timing measurements as in FuzzyTime [32] and TimeWarp [57]. These techniques can only defeat attacks which rely on measuring access latency, but cannot prevent other attacks such as alias-driven attacks [28]. They compromise the precision of the clock for the remaining workload, thus affecting functionality requirements.

RPCache [73] randomizes the mapping of all memory lines of a protected application at a per-set granularity from their actual cache set to a randomly mapped cache set, by using a permutation table. NewCache [53] randomizes the mapping at a per-line granularity using a Random Mapping Table. Both RPCache and NewCache schemes do not scale well with the number of lines in the cache (not applicable for larger LLCs) and the number of protected domains. Random Fill Cache [52] mitigates only reuse-based cache collision attacks by replacing deterministic fetching with randomly filling the cache within a configurable neighborhood window whose size impacts the performance degradation incurred. It does not scale well with an increasing TEE size.

Time-Secure Cache [69] uses a set-associative cache indexed with a keyed function using the cache line address and Process ID as its input. However, a weak low-entropy indexing function is used, thus re-keying is frequently required followed by cache flushing which requires complex management and impacts performance. CEASER [63] also uses a keyed indexing function but without the Process ID, thus also requiring frequent re-keying of its index derivation function and re-mapping to limit the time interval for an attack. A concurrent work, ScatterCache [74], uses keyed cryptographic indexing that depends on the security domain, where cache set indexing is different and pseudo-random for every domain but consistent for any given key. Thus, re-keying may still be required at time intervals to hinder the profiling and exploitation efforts of an adversary attempting to construct and use an eviction set to collide with the victim access of interest. HYBCACHE, on the other hand, leverages randomization by disabling set-associativity altogether and using random replacement for isolated execution. Every given memory address can be cached in any of the available *subcache* ways and placement is random and unpredictable; it varies randomly every time the same memory line is brought in cache.

9 Conclusion

In this paper, we proposed a generic mechanism for flexible and "soft" partitioning of set-associative memory structures and applied it to multi-core caches, which we call HYBCACHE. HYBCACHE effectively thwarts contention-based and access-based cache attacks by selectively applying side-channel-resilient cache behavior only for code in isolated execution domains (e.g., TEEs). Meanwhile, non-isolated execution continues to utilize unaltered and conventional cache behavior, capacity and performance. This addresses the persistent performance/security trade-off with caches by providing the additional side-channel resilience guarantee, and the resulting performance degradation, only for the security-critical execution subset of the workload (usually isolated in a TEE) by eliminating the fundamental causes of these attacks. We evaluated HYBCACHE with the SPEC CPU2006 benchmark and show a performance overhead of up to 5% for isolated execution and no overhead for the non-isolated execution.

Acknowledgments

We thank our anonymous reviewers for their valuable and constructive feedback. We also acknowledge the relevant work of Tassneem Helal during her bachelor's thesis. This work was supported by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), the German Research Foundation (DFG) through CRC 1119 CROSSING P3, and the German Federal Ministry of Education and Research through CRISP.

References

[1] INTEL. Intel Xeon Processors. https://www.intel.com/content/www/us/en/products/processors/xeon.html, 2009.

[2] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, 2016.

[3] INTEL. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, 2016.

[4] Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/

01/reading-privileged-memory-with-side.html, 2018.

[5] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. *ACM Symposium on Information, computer and communications security*, pages 312–320, 2007.

[6] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. *Cryptographers' Track at the RSA Conference*, pages 225–242, 2007.

[7] ARM Limited. ARM Security Technology – Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[8] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2), 2011.

[10] Joseph Bonneau and Ilya Mironov. Cache-collision Timing Attacks Against AES. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2006.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2017.

[12] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-step. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution - SysTEX'17*. ACM Press, 2017.

[13] Victor Costan and Srinivas Devadas. Intel SGX Explained. Technical report, Cryptology ePrint Archive. Report 2016/086, 2016. https://eprint.iacr.org/2016/086.pdf.

[14] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[15] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-free High-precision L3 Cache Attack Using Intel TSX. In *USENIX Security Symposium*, 2017.

[16] Goran Doychev and Boris Köpf. Rigorous Analysis of Software Countermeasures Against Cache Attacks. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2017.

[17] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*. ACM, 2013.

[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[19] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707, 2018.

[20] Michael Godfrey. On The Prevention of Cache-Based Side-Channel Attacks in a Cloud Environment. Master's thesis, Queen's University, Ontario, Canada, 2013.

[21] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*, 2017.

[22] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[23] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[24] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-channel Protection Using Hardware Transactional Memory. In *USENIX Security Symposium*. USENIX Association, 2017.

[25] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[26] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer-Verlag, 2016.

[27] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[28] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2016.

[29] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2011.

[30] Xiaochen Guo, Engin Ipek, and Tolga Soyata. Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2010.

[31] F. Hameed, A. A. Khan, and J. Castrillon. Performance and Energy-Efficient Design of STT-RAM Last-Level Cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2018.

[32] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1991.

[33] Intel. Intel Software Guard Extensions. Tutorial slides. https://software.intel.com/sites/default/files/332680-002.pdf. Reference Number: 332680-002, revision 1.1.

[34] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2015.

[35] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2016.

[36] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 151–162, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive Insertion Policies for Managing Shared Caches. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2008.

[38] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A High-resolution Side-channel Attack on Last-level Cache. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2016.

[39] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. RIC: Relaxed Inclusion Caches for mitigating LLC side-channel attacks. In *IEEE/ACM Design Automation Conference (DAC)*, 2017.

[40] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*. USENIX Association, 2012.

[41] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[42] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[43] Helge Klein. Modern multi-process browser architecture. https://helgeklein.com/blog/2019/01/modern-multi-process-browser-architecture/, 2019.

[44] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203*, 2018.

[45] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic Quantification of Cache Side-channels. In *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2012.

[46] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Security Symposium*, 2018.

[47] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. *USENIX Security Symposium*, pages 16–18, 2017.

[48] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee. A Fully Associative, Tagless DRAM Cache. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2015.

[49] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[51] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[52] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014.

[53] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[54] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2015.

[55] Wanli Liu and Donald Yeung. Using Aggressor Thread Information to Improve Shared Cache Management for CMPs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[56] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[57] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2012.

[58] Matt Miller. Mitigating arbitrary native code execution in microsoft edge. https://blogs.windows.com/msedgedev/2017/02/23/mitigating-/, Jun 2018.

[59] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. *Cryptographers' Track at the RSA Conference*, pages 21–44, 2018. 10.1007/978-3-319-76953-0_2.

[60] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. Technical report, arXiv:1703.06986 [cs.CR], 2017. https://arxiv.org/abs/1703.06986.

[61] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[62] Pulp-Platform. Ariane RISC-V CPU. https://github.com/pulp-platform/ariane.

[63] Moinuddin K. Qureshi. Ceaser: Mitigating Conflict-based Cache Attacks via Encrypted-Address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[64] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006.

[65] Daniel Sanchez and Christos Kozyrakis. Scalable and Efficient Fine-Grained Cache Partitioning with Vantage. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[66] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[67] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. *CoRR*, abs/1811.07153, 2018.

[68] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy managemen. In *USENIX Security Symposium*, 2017.

[69] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2018.

[70] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, 2018.

[71] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium*, 2018.

[72] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2016.

[73] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2007.

[74] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[75] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

[76] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2017.

[77] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. To appear in the *Proceedings of the IEEE Symposium on Security & Privacy (IEEE S&P)*, May 2019.

[78] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*, 2014.

[79] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. volume 7, pages 99–112. Springer, 2017.

[80] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. Cryptology ePrint Archive, Report 2016/980, 2016. https://eprint.iacr.org/2016/980.

[81] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2012.

[82] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

CFInsight: A Comprehensive Metric for CFI Policies

Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi
Technical University of Darmstadt
tommaso.frassetto@trust.tu-darmstadt.de, patrick.jauernig@trust.tu-darmstadt.de,
david.koisser@trust.tu-darmstadt.de, ahmad.sadeghi@trust.tu-darmstadt.de

***Abstract*—Software vulnerabilities are one of the major threats to computer security and have caused substantial damage over the past decades. Consequently, numerous techniques have been proposed to mitigate the risk of exploitation of vulnerable programs. One of the most relevant defense mechanisms is Control-Flow Integrity (CFI): multiple variants have been introduced and extensively discussed in academia as well as deployed in the industry. However, it is hard to compare the security guarantees of these implementations as existing metrics (such as AIR) do not consider the different usefulness to the attacker of different basic blocks, which are the fundamental components that constitute the code of any application.**

This paper introduces BlockInsulation and CFGInsulation, novel metrics designed to overcome this limitation by modeling the usefulness of basic blocks for an attacker trying to traverse the program's control-flow graph. Moreover, we propose a new CFI policy generator, named NumCFI, which is orthogonal to existing policy generators and prevents the attacker from taking shortcuts from vulnerable code to a system call instruction. We evaluate NumCFI, as well as a number of other CFI policy generators, using BlockInsulation, CFGInsulation, and existing metrics. Lastly, we describe L+TCFI, our implementation that combines NumCFI and an existing label-based policy, with a performance overhead of just 1.27%.

I. Introduction

Since their invention, computer systems have become responsible for increasingly complex tasks. As a result, computer programs have become increasingly complex as well. Due to this complexity and the presence of legacy code, most modern software projects are plagued by several security vulnerabilities. A number of approaches have been suggested to find these security vulnerabilities, including software testing, fuzzing, and formal methods. However, proving that software is free from vulnerabilities is only feasible for small programs. Thus, researchers have proposed a number of strategies that aim to mitigate vulnerabilities in running programs.

These run-time mitigations are usually based on one of two principles: either preventing the adversary from learning some information that is necessary to perform an attack, or inserting additional checks in the program to make an attack impossible. An example of the former, which is currently deployed in most operating systems, is Address Space Layout Randomization (ASLR) [37], which randomizes the memory layout of a program and hides it from the adversary. An example of the latter, also widely adopted in the software industry, is Control-Flow Integrity (CFI) [2], which ensures that a part of a program can only transfer control to a different part if this transfer was intended by the programmer.

This paper is focused on CFI, whose main idea is to add checks to all *indirect control flow transfer instructions*, i.e., machine instructions that transfer control to a dynamically-computed address. CFI allows each of these instructions to only transfer control to a subset of targets, according to a *control-flow graph* (CFG). Since its introduction by Abadi et al. in 2005 [2], CFI has been the focus of a large corpus of research works. There are many variants with different granularity and based on either hardware or software. Given the number of different approaches, it is important to be able to compare them in terms of both performance overhead and effectiveness against memory-corruption attacks. While there is a widely accepted metric to compare their performance overhead, i.e., the run time overhead of the execution of a standard benchmark, there is no single metric that is widely recognized by the community to compare the security protection provided by different approaches. The best-known metric is Average Indirect-target Reduction (AIR) [55], which is defined as the average reduction of allowed targets across every indirect control flow transfer instruction (the higher the better). However, AIR is not a good metric to compare different policies [51], as most CFI papers that rely on AIR report similar values greater than 99% [6]. Yet, even implementations with very high AIR are still vulnerable [13], [21]: in other words, missing even less than 1 percentage point in AIR is enough to perform attacks. Hence, AIR is not a good instrument to distinguish between and compare CFI approaches. After AIR, a number of other metrics, including AIA [18], QuantitativeSecurity [6], and CTR [34], have been proposed, as we discuss in Section IX. However, they all share a major shortcoming: they do not consider the usefulness of different basic blocks to construct an attack, instead only considering their quantity. Hence, there is a need for a new approach that leverages not only local information regarding single basic blocks, but also their position and connectivity in the full CFG. In this paper, we introduce CFInsight, a new CFI evaluation methodology and framework that achieves that.

CFInsight. A very important building block for a run-time attack is the possibility to invoke a system call with controlled parameters. This is useful, e.g., to start a new

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23165
www.ndss-symposium.org

malicious process or to change the memory protection settings. Performing a system call is also the only way to exfiltrate files or further compromise the machine, and is used in real-world exploits [10]. Hence, our CFI evaluation framework CFInsight is based on the assumption that the attacker found a vulnerable basic block and wants to perform a controlled system call. Usually, many blocks containing system call instructions exist in the program: some are unreachable from the vulnerable block, while others can be reached using a number of different paths through the CFG. We construct a novel metric, CFGInsulation, which considers the number and length of these paths from a basic block to a system call, and quantifies how easy it is for an attacker to build an exploit. We model a number of CFI policy generators: a theoretical perfect one, generators based on matching function types or number of arguments, and a generator that allows transfers to any valid function. We show that we can apply CFInsight to the generated policies and compute their CFGInsulation to compare them, showing how CFGInsulation allows to distinguish between policies with very similar AIR.

NumCFI. Moreover, we leverage the data generated by CFInsight to define and evaluate a new CFI policy generator, dubbed NumCFI. NumCFI assigns each basic block a *tag*, which is the length of the shortest path from the block to a system call instruction, and it enforces the property that a block with tag t can only call blocks with tag $\geq t-1$. In other words, any attack that starts in a basic block and requires a system call needs to go through as many basic blocks as the shortest legal path from the starting node to a system call instruction; the attacker cannot "take shortcuts," but has to go through the specified number of basic blocks instead. We show that NumCFI has a comparable or better CFGInsulation than a type-based policy generator, and that combining them leads to significant improvements over either one. We demonstrate that this combination is practical with a prototype implementation, which we call L+TCFI, and show that it has a very low run-time overhead (1.27% on benchmarks of the SPEC CPU2017 suite).

Contributions. In this paper we make the following contributions:

- We describe, design, and implement a novel CFI evaluation framework, CFInsight, based on measuring expressive properties of the CFGs of real programs, instead of simply counting reachable basic blocks. We plan to open source CFInsight so it can be useful to the community.
- We apply CFInsight to better compare the relative security characteristics of multiple state-of-the-art CFI policy generators, using our new CFI metric, CFGInsulation. We compare CFGInsulation with four existing CFI metrics.
- We leverage the knowledge generated by CFInsight to define a new CFI policy generator, NumCFI, and show that it significantly improves the security guarantees of other widely used CFI policy generators.
- We design a generic CFI implementation, L+TCFI, which can be used to enforce a combination of NumCFI with a classic label-based CFI, and we show that it has a very low run-time overhead (1.27% on benchmarks of the SPEC CPU2017 suite).

The rest of the paper is organized as follows: Section II introduces a number of topics that are required to understand the rest of the paper; Section III describes our approach and our metrics; Section IV describes our analyzer, which computes these metrics; Section V applies the analyzer to a number of existing CFI policy generators and discusses the resulting metrics; Section VI describes NumCFI; Section VII adds NumCFI to our analysis; Section VIII describes and evaluates our CFI implementation L+TCFI; Section IX discusses related works and Section X concludes the paper.

II. Background

This section introduces control-flow graphs, run-time attacks and control-flow integrity.

A. Control-Flow Graphs

A *control-flow graph* (CFG) is a directed graph representing the control flow of a program. It consists of nodes, which represent the basic blocks in the program, and edges representing legal transitions from one basic block to another. A basic block is a contiguous sequence of instructions that does not have any internal branch: branch instructions can only be the last instruction of a basic block, and instructions targeted by a branch can only be the first instruction of a basic block. CFGs (and the basic blocks they contain) are a popular abstraction used to analyze computer programs. However, generating CFGs is not trivial. They can be generated either statically or dynamically. Static generation leverages compiler passes (or an equivalent for binaries) to decide based on the observed instruction whether a new basic block is formed or if there is a transition from one basic block to another. These transitions are caused by branches. Determining all the possible destinations of a branch is hard in practice, as a common construct used in programs are indirect jumps. Indirect jumps get their target from a register, hence, this target cannot be resolved statically in the general case, but only approximated. While modern techniques like symbolic execution can help to solve this problem, the generated CFG is still an approximation in practice. In contrast, dynamic approaches monitor the behavior of the program at run time. Hardware features like Intel PT or debugging functionality allow to extract the actual targets of indirect jumps. Nonetheless, this approach also cannot fully solve the problem, as dynamic approaches can only monitor the control flow for taken branches. Since the information observed depends on the program's input, a large set of inputs might be needed to generate a close-to-perfect CFG, which can be achieved through the use of automated testing (fuzzing) or a test suite.

B. Run-time Attacks

Run-time attacks have been a persistent threat for modern computing platforms for more than three decades. These attacks exploit vulnerabilities in software to achieve arbitrary code execution. Memory corruption attacks have a long-standing history. The very first attacks exploited buffer overflows in memory to inject new code into the data section and execute it later, which effectively added a new node to the CFG. However, these attacks were still primitive, and easy to mitigate. By introducing a write-xor-execute (W$\oplus$X) policy, attackers could no longer inject executable data, stopping

code injection attacks altogether. This mitigation has been deployed broadly, and is most prominently known as Data Execution Prevention (DEP). Although this mitigation raised the bar, attackers found new strategies to bypass these defenses using more sophisticated attacks that do not add nodes to the CFG, but add new paths between existing nodes, hence called code-reuse attacks. Code-reuse attacks can be categorized into full-function reuse attacks (e.g., return-to-libc [49]) and return-oriented programming (ROP) [45], [9]. ROP uses small sequences of instructions to form gadgets, which can be used as building blocks to mount a more complex attack or achieve Turing-complete computation. While simple defenses like Address-Space Layout Randomization (ASLR) were deployed in real systems, ROP remains challenging to prevent, especially since code-reuse attacks can be combined with information leakage (e.g., the JIT-ROP attack [48]). These enhanced attacks spawned advanced defenses both in hardware and software. Prominent examples of defenses are Control-Flow Integrity (CFI) [2], [12], [19], [3], Code-Pointer Integrity (CPI) [28], or sophisticated randomization techniques [11], [48]. Some defenses are already deployed in products, e.g., Microsoft's Control-Flow Guard (CFGuard), Clang's CFI [30] which is used in Google Chrome, Intel's Control-flow Enforcement Technology (CET) [25] and ARM's Pointer Authentication (PAC) [41]. Due to the progressive adoption of some of these defenses, a more advanced type of attack has been introduced in the academic world. In a Data-oriented Programming (DOP) attack [24], [26], non-control data is manipulated to reuse valid paths under CFI to achieve Turing-complete computation. While schemes like Data-Flow Integrity [8], [50] solve this theoretically, they come at a significant performance and hardware overhead. As a result, solving this problem remains challenging in practice.

C. Control-Flow Integrity

Control-Flow Integrity relies on the fact that most functions in a program only call a very limited subset of the other functions. Given a CFG of a program, a CFI implementation instruments the code such that only these transfers are allowed, and any attempt to deviate from the CFG is detected. Only function calls that compute their target at run time, i.e., *indirect* function calls, are potentially vulnerable and need to be instrumented; direct function calls have a hard-coded target that cannot be changed at run time. A *context-insensitive* CFI policy specifies, for every indirect function call site, which other functions can legitimately be called from that site. A *context-sensitive* CFI policy considers not only the identity of the call site and the callee, but also other criteria, like the value of a variable or the top of the call stack, in order to decide whether an indirect call is legal. Moreover, the call to a function (forward edge) is not the only one that needs to be protected, the return (backward edge) needs it too [7], e.g., in the form of a shadow stack [5], a data structure keeping secure copies of return addresses.

Deploying CFI poses a number of challenges. One such challenge is *overapproximation* of the allowed control flow transfers, which is mostly introduced in the name of performance. A precise run-time instrumentation needs to check if a specific target is allowed for the specific caller, which can be relatively slow. Thus, most *CFI policy generators* introduce overapproximations in order to streamline the checks and make

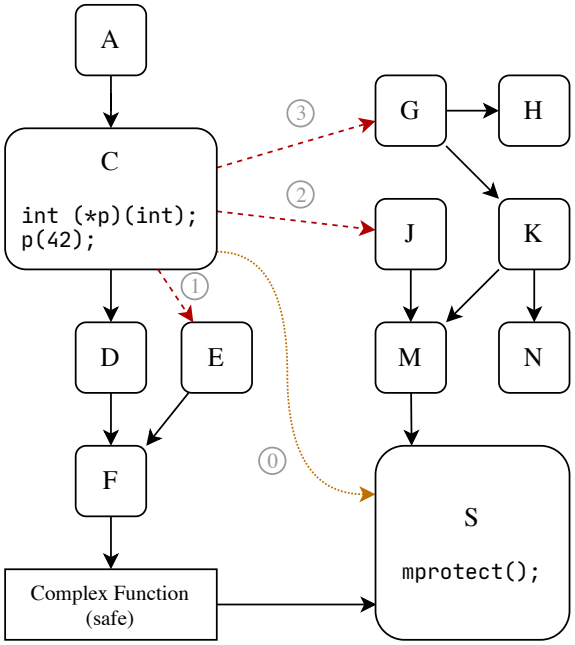

Fig. 1. A CFG for a simple program with a corruptible pointer.

them faster. A common method to simplify CFI checks is to assign a single numeric *label* to every indirect caller and callee and check that the label of the caller matches that of the callee. This effectively splits the nodes into *equivalence classes*, one for each label, and allows the run-time check to be a simple (and fast) integer comparison. As an example, a CFI implementation can label functions according to the return type and type of the parameters [38]. However, it introduces overapproximation because unrelated nodes need to have the same label in order for the scheme to work, i.e., it is not possible to distinguish between targets in the same equivalence class. This overapproximation has been shown to be sufficient to attack protected programs [13], [17]. In practice, most of the deployed CFI implementations use either type-based policies [30] or simple heuristics, like checking whether the callee address is the beginning of a function [32].

A recent trend is to add hardware support for CFI [25], [29], improving performance and providing better integrity protection for the CFI mechanism itself.

III. CFInsight

We begin our description of CFInsight by looking at the sample CFG in Figure 1. In this program, basic block C contains a function pointer `p` which can be corrupted by the adversary. The adversary can leverage this vulnerability to launch a code-reuse attack, and wants to reach block S, which contains an invocation to the system call `mprotect`. If the adversary can reach this block and control the parameters to the system call, it is trivial to disable memory protection and then perform a classic code injection attack. If the program is not protected by CFI, the adversary can simply redirect the control flow to block S using edge ⓪ and then perform the rest of the attack. However, if the program is protected by CFI, only CFG edges allowed by the *CFI policy* can be followed;

as a result, the attacker is limited to these allowed edges. In the case of a perfect CFI policy (which only allows black solid edges from Figure 1), the only path to S goes through a complex function, which we assume to be implemented with secure programming techniques: as a result, it is likely that the attacker can only invoke the system call using safe parameters and, hence, cannot launch the attack.

Real-world CFI implementations, however, are not perfect, as we mention in Section II-C, and often use *overapproximated policies*, i.e., they allow edges that should be forbidden. The impact of this overapproximation on the security of the program depends on which illegal edge is incorrectly allowed. As an example, if edge ① is allowed, the adversary gains no advantage, since the only path to S still goes through the safe function. If one of the edges ② and ③ is included, the attacker can instead jump to J or G and follow more nodes until the control flow reaches S.

From the perspective of existing metrics, like AIR [55], AIA [18], or CTR [34], a policy that includes edge ① is equivalent to one that includes edge ②, or one that includes ③, as they allow the same number of edges starting from C. However, they are not equivalent in practice. If only ① is allowed, the attacker has no advantage over a perfect CFI, since there is no additional path to S. If only ② is allowed, the attacker has a substantial advantage: the attacker can jump to J and then follow the flow to M and S. Lastly, if only ③ is allowed, the attacker still has an advantage, but smaller than the previous case: The attacker must jump to G and try to follow the chain all the way to S. In order for this to be successful, the adversary needs to ensure that the desired branch to K is taken in G, instead of the branch to H (similarly in K, with the branch to M). Which branch is taken depends on a condition, which could be out of the attacker's control.

The purpose of CFInsight is to compare CFI policies considering their graph structure and connectivity. We focus on context-insensitive CFI policies, since most CFI policies deployed in practice fall in this category [2], [30], [25]; however, our approach can also be applied to context-sensitive CFI policies, as we discuss in Section IV-D. For each indirect function call, we measure the quantity and length of possible paths that lead to a system call instruction.

In the following, we first describe our threat model, then we explain how our metric is defined and how we compute it.

A. Threat Model and Assumptions

With CFInsight we aim to model how most run-time attacks start in the real world. Thus, we make the following assumptions about the victim program and the capabilities of the adversary:

- **A0** The adversary wants to attack a vulnerable program. More concretely, the goal of the adversary is to invoke a system call with controlled parameters, e.g., to start a new malicious process or to change the memory protection settings. Performing a system call is the only way to exfiltrate files or further compromise the machine, and is used in real-world exploits [10].
- **A1** The adversary has access to a vulnerability in the program that allows arbitrary read operations to readable memory and arbitrary write operations to writable memory.
- **A2** The adversary can leverage the arbitrary write primitive to corrupt the memory such that an indirect call will be redirected to an unintended target. As an example, this can be done in the presence of a buffer overflow vulnerability. The adversary can corrupt pointers and hijack the control flow multiple times. If a CFI policy is in place, all of the hijacked calls need to comply with the CFI policy.
- **A3** We assume W⊕X (see Section II-B) to be in place and working, i.e., the adversary cannot overwrite the application code or inject new code.
- **A4** We assume that a shadow stack implementation [5], or equivalent, is deployed on the victim, hence, the attacker cannot target the function returns. Protecting function returns is a very different problem than protecting function calls, and this paper is focused on the latter.
- **A5** We assume the adversary to be able to bypass any randomization-based defense in use, e.g., ASLR; thus, we do not consider them in our model.
- **A6** In principle, our approach can be applied to any operating system. However, a number of low-level details differ between them. Hence, we focus on Linux, in line with related work [6], [7], [16], [17].
- **A7** We expect the victim program to be built using the current best practices for Linux software, e.g., full RELRO [47], which makes the Procedure Linkage Table (PLT) read-only. Thus, the attacker cannot overwrite PLT entries.

B. Our Observations: Single-Node Metric

In CFInsight, we aim to define quantifiable properties of a graph that measure how easily an attacker can build a successful attack. We begin by considering a given node in the CFG that calls a vulnerable code pointer, and a specific system call site the adversary needs to reach. To reach this goal, the adversary needs to follow a number of CFG edges, which need to be legal according to the current CFI policy. Let us consider one such path. Each basic block on this path contains machine instructions, which perform a number of operations, and ends with a (possibly conditional) branch instruction. As a result, traversing each basic block poses two challenges for the adversary. First, if the branch is conditional, the adversary needs to make sure the value of the branch condition is true if the branch is to be taken, or false otherwise. Second, the code in the basic block often writes data to memory or to a register; this might overwrite some data the adversary prepared for the attack, e.g., a parameter of the system call or the operand of a branch condition. Our first observation follows:

- **O1** The more basic blocks an attack needs to traverse, the harder the attack is.

However, there usually are multiple paths between a node and a system call site. The attacker only needs one path that supports an attack, and hence:

- **O2** The more paths are available for an attack, the higher the likelihood that at least one of them is viable for the attack.

We leverage these observations to build our metric to measure the effectiveness of CFI policies. As a first approximation, our metric is the ratio between the length of paths to any system call, and the number of these paths. Our metric is directly proportional to the length of the paths, due to

Observation O1, and inversely proportional to their number, due to Observation O2; higher values of the metric indicate that the attack is harder. However, this approximation needs to be refined to be applicable in practice. First, there are multiple paths of varying lengths starting in a given node and ending in some system call site; since it is not computationally feasible to examine all paths in a complex CFG, we consider instead the lower bound of their lengths, i.e., the shortest path from the node to any system call. Second, it is also infeasible to know the exact number of paths from a given node to a system call; a useful approximation is to consider the number of *linearly independent* paths, which can be computed efficiently[1].

The result is our metric that quantifies the difficulty of an attack starting in a basic block b and reaching any system call site. We call this metric BLOCKINSULATION(b) and we define it as:

$$\frac{\text{length of shortest path } b \rightarrow syscall}{\text{№ linearly independent paths } b \rightarrow syscall}$$

If there is no path between b and any system call site, we define BLOCKINSULATION(b) $= \infty$, since any attack is impossible in this case.

C. Whole-Program Metric

In general, considering the whole distribution of values of BLOCKINSULATION of all basic blocks gives the most complete picture. However, it can also be useful to define a single numeric metric to summarize the distribution of the BLOCKINSULATION. Simply averaging the values is impractical, since the metric we defined can assume values from ≈ 0 to ∞. We instead decide to take the median value of the distribution, which is often a finite value. If more than half of the values are ∞ and the median is infinite, we instead take the maximum finite value. A greater value of CFGINSULATION indicates a program that is harder for an attacker to exploit.

CFINSIGHT leverages this new metric to compare the security guarantees of different CFI policy generators.

IV. CFINSIGHT ANALYZER

In the previous section we introduced a metric to measure the security guarantees of a CFI policy. In Figure 2 we show the overall design of the analyzer we designed to compute this metric. Each component is described in detail below.

A. CFG Generation

In order to compute our metric for a program we need its CFG. As we explain in Section II-A, there are two main approaches to generate a CFG: either through dynamic or static analysis. Both approaches have different advantages and limitations. While we consider the problem of enhancing CFG generation techniques to be orthogonal to the scope of this paper, the quality of our analysis does depend on the quality of the CFG it uses. Hence, we leverage both approaches: we trace a number of executions of the program on a set of inputs and we also perform a static analysis of the program.

[1]The number of linearly independent paths in a graph, also known as McCabe's cyclomatic complexity [31], can be easily computed as $|E| - |N| + 2$, where $|E|$ is the number of edges and $|N|$ the number of nodes of the graph.

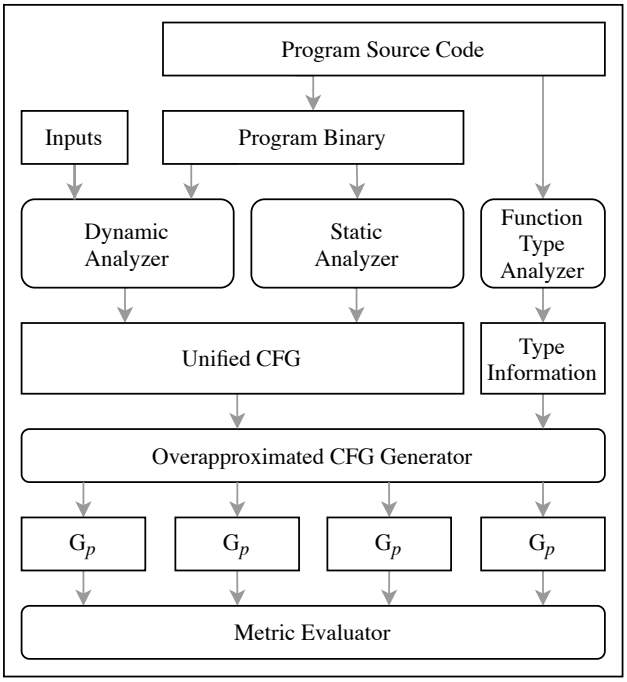

Fig. 2. Architecture of the CFINSIGHT analyzer.

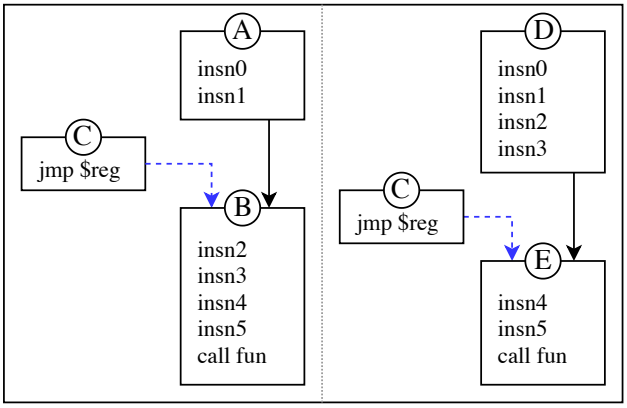

Fig. 3. Depending on the target of the jump instruction in block C (dashed blue edge), two different pairs of basic blocks are generated: A+B or D+E.

We then combine the CFGs generated by these tools into a single unified CFG for the program under analysis. Combining multiple CFGs is not trivial, since different tools can split a binary into different sets of basic blocks. As an example, Figure 3 shows two different CFGs generated for the same program. Assume that the jump instruction in block C can legally jump to either *insn2* or *insn4*, and that different tools generate CFGs which only contain one of these edges each. As a result of the different edges, in the CFG on the left the code is split resulting in blocks A and B, while in the CFG on the right the split results in blocks D and E.

While it is easy to determine whether an instruction terminates a basic block (any branch instruction does), determining where a basic block starts is more complex. By definition, a basic block is a sequence of instructions that will be always executed one after another. Since the tools often generate dif-

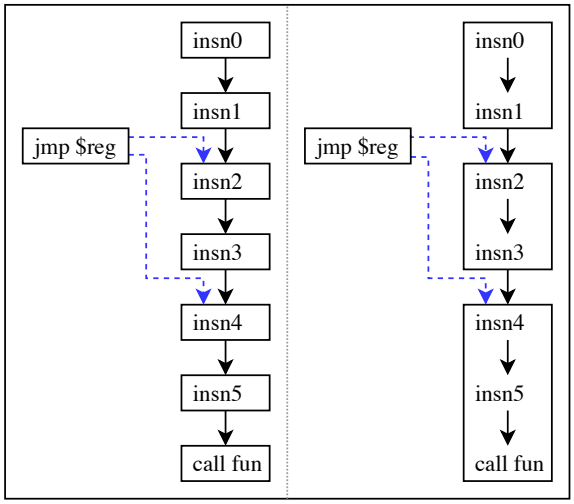

Fig. 4. Merging the graphs of Figure 3. Left: decomposition of the basic blocks in single instructions. Right: recomposed basic blocks.

ferent sets of edges, and since the way instructions are grouped into blocks depends on the known edges, different tools (and concrete executions on different inputs) often produce different basic block sets for the same instructions.

We address the problem by splitting all basic blocks into instructions, building a list of all edges on the instruction level (see left side of Figure 4). We then find the new basic block boundaries based on all known edges for the program (right side of Figure 4), thus generating a *unified CFG*.

In order to make the next steps more straightforward, we add an additional node to the CFG, called *target*, which symbolizes the attacker's goal. In our attacker model (Assumption A0), the adversary's goal is to reach a system call, so we add an edge from any block containing a system call to *target*.

B. Overapproximating the CFG

Once we obtain a unified CFG for a program, we need to model the effect of the various CFI polices on the program. As we mentioned in Section II-C, CFI policy generators introduce overapproximations in order to simplify their construction or improve their performance. For each generated CFI policy p, we model its effect on a program's CFG and generate a list of edges that are not present in the CFG but are allowed by the policy. We then add these edges to the original CFG and generate G^p, i.e., the graph of all allowed control flow transfers under policy p.

We model a number of CFI policy generators used in widely deployed mitigation schemes, as well as the lack of CFI:

SoFCFI a CFI policy generator that allows all functions to be the target of any indirect function call, similarly to what can be done with Intel CET [25];

TypeCFI a CFI policy generator that only allows indirect function calls if the type signature of the callee matches the type expected at the caller side, like RAP [38];

NumArgCFI a simplified variant of TypeCFI, which only checks that the number of arguments of the callee matches the number provided at the caller side. This policy generator is an idealized version of TypeArmor [53];

NoCFI the absence of CFI can be modeled by a policy that allows each indirect call site to call any other basic block[2] in the binary.

In order to compute SoFCFI, we only need the addresses of the functions, which we can extract from the symbols in the binary. For TypeCFI and NumArgCFI, we additionally need type information. We extract the types of functions from the debug symbols, while we leverage a custom compiler pass to extract the expected function type at the indirect call sites.

C. Computing Our Metric

Once we generate the overapproximated graph G^p for a policy p, we can use it to compute BLOCKINSULATION (see Section III-B). The first step is to compute, for every indirect call site b, the subgraph containing all paths to *target* (see Section IV-A). Since we are only interested in nodes that have a path to *target*, i.e., its *ancestors*, we focus only on them for efficiency reasons. We compute the set of ancestors by performing a depth-first search, starting in *target*, in a copy of the graph where all edges are reversed. Afterwards, for each indirect call site b, we perform a depth-first search, only considering the ancestor nodes we found before. The nodes found in this search compose the subgraph we wanted to build. After this subgraph is known, our metric can be computed in a straightforward manner.

The naive representation of the G^p graphs in memory is challenging, since naively representing some CFI policies requires a large number of edges. As an example, in NoCFI, any indirect call site can transfer control to any basic block (see Section IV-B), which produces $|I| \times |N|$ edges for a CFG with $|I|$ indirect call sites and $|N|$ basic blocks. In order to produce a more tractable representation of this graph, we introduce a synthetic node, called *any*. We then create an edge $i \rightarrow any$ for every $i \in I$, and an edge $any \rightarrow b$ for every $b \in N$. This leads to a graph with the same connectivity, with only $|I| + |N|$ edges, which is a substantially lower number. However, if not accounted for, this optimization would lead to different result for our metric. Hence, while computing our metric in the presence of synthetic nodes, we take this difference into account, in order to compute the value the metric would have[3] in the naive version of the graph.

D. Extensions and Discussion

The framework is built in a modular way and it can easily be extended. For example, an analyst can add an additional CFG generator, mark further blocks as the attacker's target, or model a new CFI policy generator. In particular, CFINSIGHT can also be extended to consider a context-sensitive CFI policy (see Section II-C). Representing a context-sensitive CFI policy

[2] In a variable-length instruction set like x86, in the absence of CFI, the attacker can also jump in between instructions; we do not consider this in our model, since the adversary does not need this possibility to very easily reach a system call (without CFI).

[3] A synthetic node with i incoming and j outgoing edges represents the fact that each of these i predecessors can reach any of the j successors. As a result, these $i + j$ edges in the graph actually represent $i \times j$ edges. We then adjust the edge count by adding $i \times j - (i + j)$ and the node count by subtracting one.

requires having multiple nodes in the CFG for the same basic block, one for each context. Our methods can then be applied to this extended CFG.

In the presence of a multi-threaded application, CFInsight considers each thread separately. As a result, it does not directly model an attack where two or more threads are exploited at the same time and collaborate to perform a system call. However, in this case, we focus on the thread that performs the system call. Its control flow needs to reach a system call site, starting from a legitimate block; as a result, our analysis still applies.

V. CFInsight: Implementation and Results

In this Section we describe our CFInsight implementation and we present its results.

A. Analyzer: Implementation Notes

We implemented our prototype of the CFInsight analyzer as a number of Python scripts, totaling approximately 4 000 lines of code.

We compile all binaries with the Clang compiler (version 11.0.1), which we extend with a custom IR pass to produce a list of expected function types at indirect call sites (see Section IV-B). For our static analysis we use the angr framework [46], which can generate the CFG of a program using static analysis and symbolic execution. For our dynamic analysis we choose CFGgrind [43], a Valgrind-based tool that dynamically records control flow transitions as they happen during program execution. In angr, we generate both a *fast* and an *emulated* CFG, while in CFGgrind we generate a separate CFG for every input file; all of them are then combined into a unified CFG, like we explain in Section IV-A. We extract the function types for TypeCFI from DWARF debug symbols, which encode the types of the functions (together with other information) in the binary itself. We decode this data in our DWARF parser, which is based on pyelftools [4]. We also retrieve the detached debug symbols for the system libraries from the Debian package manager, then we decode them with our DWARF parser.

The most processing-intensive part of the analysis pipeline is the evaluation of the metrics on an overapproximated graph. Since this evaluation is mostly independent for each basic block, we split the work between multiple threads (up to the number of CPU cores available), while the main thread is responsible for collecting the results.

B. Results

In this section, we report the results of our CFInsight framework on state-of-the-art CFI policy generators. In Section VII, we compare these metrics with our novel CFI policy generator NumCFI as well.

Experimental setup. In order to test CFInsight and to compare different CFI policy generators, we compute our metrics for a number of benchmarks. From SPEC CPU2017, the most recent version of a widely used benchmarking suite, we select all benchmarks written in C, C++, or a mix of the two, in their *speed* variant. For each benchmark, we statically generate its CFG with angr and we use CFGgrind to dynamically trace the execution; we use all input files that compose the three SPEC workloads (*test*, *train* and *refspeed*). Moreover, we prepare our own benchmark of the web server nginx [1]: as inputs we use the official nginx test suite, which is composed of 388 different configurations. We ran all tests on a machine running Debian Sid, last updated in April 2021, with a 32-core Intel Xeon Silver 4110 processor and 128 GB of RAM.

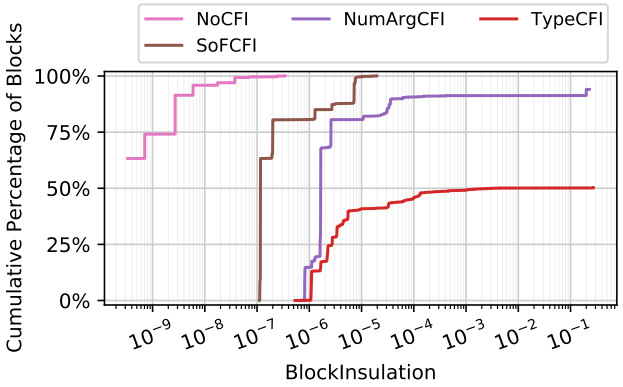

Fig. 5. CDF of the distribution of BlockInsulation of basic blocks containing indirect calls, under different CFI policies.

BlockInsulation. We evaluated BlockInsulation for every indirect call site in our benchmarks under four CFI policy generators: NoCFI, SoFCFI, NumArgCFI, and TypeCFI (in increasing order of strictness; we define them in Section IV-B). In Figure 5, we visualize the distribution of these metrics by plotting the Cumulative Distribution Function (CDF) of the BlockInsulation over all indirect call sites in all of our benchmarks[4]. Each point with coordinates $(x, y\%)$ in these curves means that $y\%$ of the blocks have a BlockInsulation $\leq x$. Since greater values of BlockInsulation indicate that attacks are harder to perform, a curve that is lower and to the right of the figure indicates a more secure CFI policy generator. As expected, the least secure policy generator is NoCFI, followed by SoFCFI, NumArgCFI, and lastly TypeCFI.

CFGInsulation. While we stress that a CDF of BlockInsulation (like Figure 5) is the most complete way to compare different policies, it is often useful to summarize the results into a simpler numeric metric, which we introduce in Section III-C. We define CFGInsulation as the median BlockInsulation value for the indirect call sites of a program (or the maximum finite value if the median is infinity). Figure 6 shows the values of this metric for all benchmarks we consider. For all benchmarks, the CFGInsulation values are in the expected order (TypeCFI, NumArgCFI, SoFCFI, NoCFI). TypeCFI improves the CFGInsulation by 3 to 7 orders of magnitude compared to NoCFI, by 1 to 5 orders of magnitude compared to SoFCFI, and up to 3 orders of magnitude compared to NumArgCFI.

[4]The graphs only show data about the main binaries. Our model also considers the dynamic libraries, but we only use the information that can be extracted from their binary and debug symbols, since compiling libraries such as libc is a very complex process.

TABLE I. COMPARISON OF CFI POLICY GENERATORS USING EXISTING METRICS AND CFGINSULATION.

CFI policy generator	mean(fAIR)*	mean(fAIA)†	sum(iCTR)†	geomean(QS)*	total CFGINSULATION*
NoCFI	0.00000%	6023518.4	324855240247	0.00000020	$3.348766 \cdot 10^{-10}$
SoFCFI	99.94011%	4504.8	406122077	0.00040833	$1.154559 \cdot 10^{-07}$
NumArgCFI	99.99284%	584.6	59811668	0.00233956	$1.644561 \cdot 10^{-06}$
TypeCFI	99.99720%	228.1	29684972	0.05289177	$3.712601 \cdot 10^{-03}$

* *Higher is better.* † *Lower is better.*

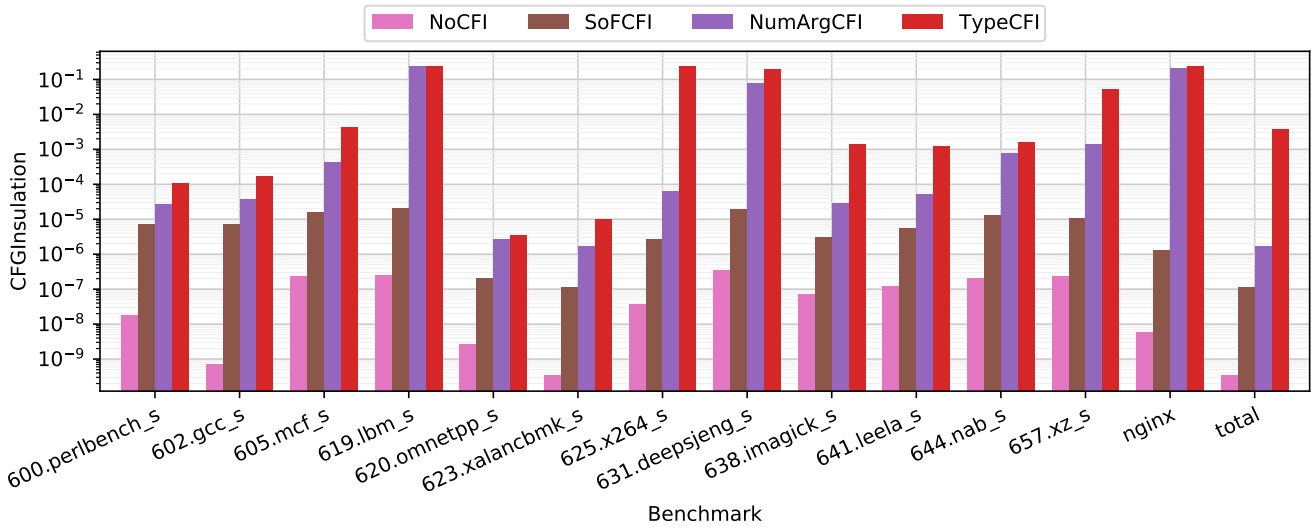

Fig. 6. CFGINSULATION for each benchmark we consider.

Other metrics. In order to validate our findings and compare the results with other existing metrics, we compute a number of CFI metrics over the same CFGs. Specifically, we compute:

- fAIR, the forward-edge variant of AIR [55]: the average reduction in the number of allowed target for every indirect function call (higher is better);
- a forward-edge variant of AIA [18] which we dub fAIA: the average number of allowed targets for every indirect function call (lower is better);
- iCTR [34]: the sum of the number of allowed targets for every indirect function call (lower is better);
- QS (QuantitativeSecurity [6]), the product of the number of equivalence classes and the inverse of the size of the largest class (higher is better).

We compute all of these metrics for each of our benchmarks. Since the metrics are defined in different ways, we use different mathematical functions to summarize them. fAIR and fAIA are defined as arithmetic means; hence, we report the arithmetic mean of the individual results from the benchmarks. iCTR is defined as a count, so we report the sum of the single results; QS is a ratio, so we report its geometric mean. These values, along with CFGINSULATION, are shown in Table I. The metrics confirm that TypeCFI offers more security than NumCFI, which is better than SoFCFI and NumArgCFI. In Section VII we extend this analysis with our novel CFI policy generators NumCFI.

VI. NUMCFI

We mentioned earlier that existing CFI metrics, like AIR, consider basic blocks with the same label equivalent to each other, leading to the division of basic blocks in equivalence classes. Our answer to this shortcoming is to propose CFINSIGHT, which analyzes a CFI-protected program in terms of how easy it is for an attacker to reach a system call instruction. The core insight is that a node that is close to a system call instruction (e.g., node J in Figure 1) is more useful to an attacker than farther nodes (e.g., node G). The same insight can be applied to produce a novel CFI policy generator as well, which led us to the definition of NumCFI.

The idea of NumCFI is to assign each basic block a *tag*, which is the number of basic blocks on the shortest path from the block to a system call instruction. As an example, Figure 7 shows the tags that NumCFI assigns to the program in Figure 1, assuming the path within Complex Function to be 10 blocks long. At run time, we enforce the property that the tag can decrease by at most 1 for every call, i.e., a block b can call a block c only if their tags t_b, t_c satisfy this property:

$$t_c \geq t_b - 1 \tag{1}$$

This prevents the attacker from "taking shortcuts" when planning an attack, i.e., if the attacker wants to hijack the control flow in a block with tag t, the attack chain needs to go through at least t blocks before it reaches a system call

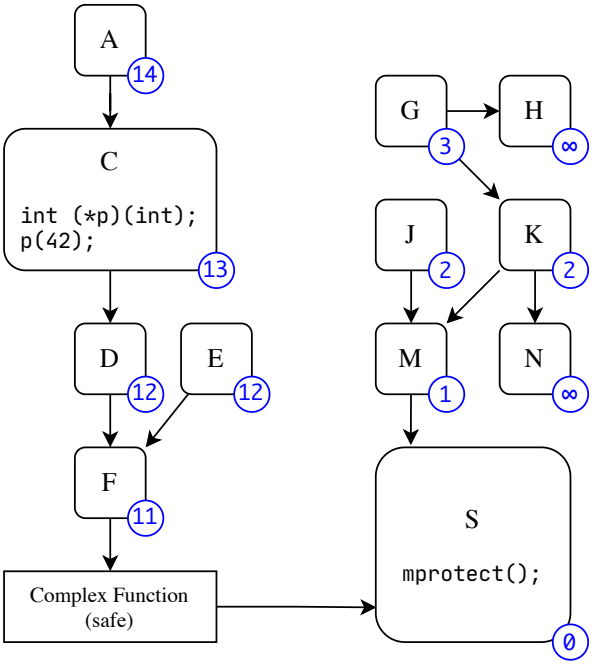

Fig. 7. The same program of Figure 1, with the NumCFI tags for each block in the blue circles. In this example we assume the path through the Complex Function to be 10 blocks long.

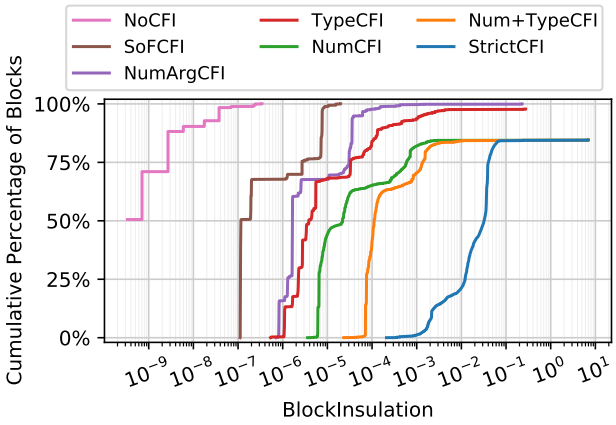

Fig. 8. CDF of the distribution of BLOCKINSULATION of basic blocks containing indirect calls, under different CFI policies.

instruction. Every basic block with no path to a system call instruction receives a special tag ∞, which only allows it to transfer control to other blocks with the tag ∞. NumCFI can also be combined with an orthogonal CFI policy generator based on labels, e.g., TypeCFI. We name this combined policy generator Num+TypeCFI.

An interesting consequence of deploying NumCFI is that it allows security analysts to focus their attention on a small number of basic blocks with a low tag value, since they are the blocks that the adversary might use to mount an attack. Blocks with higher tag values can receive less attention, since they would require long attack chains; blocks with tag ∞ can be outright ignored, since they cannot reach system call instructions at all.

Below, we discuss how NumCFI prevents the nginx attack described in [17]. We then describe how NumCFI and Num+TypeCFI compare with other policy generators in Section VII. Lastly, we write an implementation of Num+TypeCFI and we evaluate its performance overhead in Section VIII.

Case Study: Nginx. Farkhani et al. [17] construct an attack on nginx, protected by type-based CFI implementation RAP [38] The attack leverages a collision between functions with the same type. Specifically, the code of function `ngx_worker_process_exit` contains an indirect function call to a function that takes no arguments. The attack leverages this fact to hijack the control flow and call a different function, `ngx_master_process_cycle`, which also takes no arguments; from there, the control flow eventually reaches an invocation of the system call `execve`. We applied NumCFI to nginx 1.16.1 compiled for x86_64 Linux and we verified that the tags of these two basic blocks differ by more than 1, i.e., Equation (1) is not satisfied and this control flow is disallowed. Hence, NumCFI protects nginx from the attack described in [17].

A downside of all CFI implementations is that, if a valid edge is missing in its input CFG, attempting to follow that edge in the program will lead to a false-positive CFI violation. For example, the authors of RAP [38], encountered missing edges caused by incorrect type signatures in the source code of programs. The issue of missing edges also presents itself in NumCFI. The root cause of this issue is the well-known difficulty of generating a precise CFG of a program. While improving CFG generation techniques is outside of the scope of this paper, there are ways to mitigate this issue. First, one can combine multiple CFG generation approaches. In our prototype implementation of CFINSIGHT, we combine angr and CFGgrind as representatives of static and dynamic analysis tools. However, any other more advanced static analysis tool can be used as well. The dynamic analysis can be improved by increasing the number and quality of the input files. Good software engineering practices recommend the presence of a test suite which is as thorough as possible. As a result, tracing the execution of this comprehensive test suite ensures that any tested functionality is covered in the generated CFG. The coverage generated by the test suite can further be improved with other techniques such as fuzzing. Finally, if the core functionality of the program is covered by tests, any false positive that is still present is by definition only incurred in rare circumstances. These false positives can then be manually addressed just like any other rare bug.

VII. CFINSIGHT: COMPARISON OF NUMCFI

In this section, we extend the results of Section V by considering NumCFI as well. We use the same experimental setup and benchmarks we describe in Section V. Due to an imperfect CFG generation, for some indirect call sites we do not know of any legal outgoing edge: in order to ensure a fair comparison between policy generators, we omit these nodes from the following analysis.

In addition to the existing CFI policy generators (TypeCFI, NumArgCFI, SoFCFI, NoCFI), we analyze NumCFI, our new

TABLE II. COMPARISON OF NUMCFI WITH OTHER CFI POLICY GENERATORS USING EXISTING CFI METRICS AND CFGINSULATION.

CFI policy generator	mean(fAIR)*	mean(fAIA)†	sum(iCTR)†	geomean(QS)*	total CFGINSULATION*
NoCFI	0.00000% (7)	6023518.4 (7)	75153429826 (7)	0.00000020 (7)	$3.348766 \cdot 10^{-10}$ (7)
SoFCFI	99.94011% (6)	4504.8 (6)	83776616 (6)	0.00040833 (6)	$1.161153 \cdot 10^{-07}$ (6)
NumArgCFI	99.98757% (4)	1039.4 (4)	22424457 (4)	0.00197575 (4)	$1.647973 \cdot 10^{-06}$ (5)
TypeCFI	99.99498% (3)	390.6 (3)	8077681 (3)	0.04507330 (3)	$3.856191 \cdot 10^{-06}$ (4)
NumCFI	99.95866% (5)	3055.5 (5)	60548479 (5)	0.00045113 (5)	$2.184315 \cdot 10^{-05}$ (3)
Num+TypeCFI	99.99665% (2)	260.1 (2)	5615941 (2)	0.05484967 (2)	$1.109534 \cdot 10^{-04}$ (2)
StrictCFI	99.99996% (1)	2.0 (1)	10974 (1)	1.65245297 (1)	$3.225806 \cdot 10^{-02}$ (1)

** Higher is better. † Lower is better. (The number in parentheses is the rank of each CFI policy generator according to each metric.)*

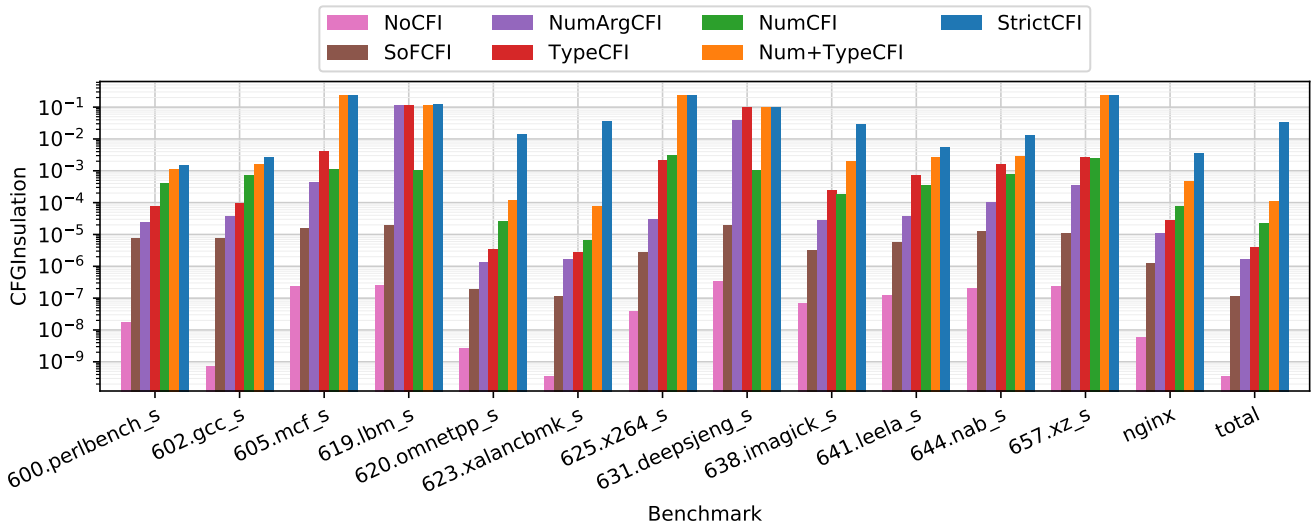

Fig. 9. CFGINSULATION for each benchmark we consider.

CFI policy generator, as well as Num+TypeCFI, which is the combination of NumCFI and TypeCFI. Finally, we define StrictCFI as a CFI policy that only allows legal edges, without any overapproximation. StrictCFI serves as an indication of the best possible context-insensitive CFI policy that can be obtained for the binaries we consider.

BLOCKINSULATION. In Figure 8, we show the cumulative distribution (CDF) of BLOCKINSULATION over the call sites. As we mentioned earlier, curves that are lower and to the right indicate more secure CFI policies. As expected, StrictCFI is the most secure policy, since it does not introduce any overapproximation. The BLOCKINSULATION of NumCFI is approximately one order of magnitude greater than TypeCFI, which is the best CFI policy generator currently deployed on a large scale. Moreover, Num+TypeCFI combines the strengths of both its components, further increasing the BLOCKINSULATION by one order of magnitude.

CFGINSULATION. As we mentioned, the most expressive way to compare two policies with the help of CFINSIGHT, is to look at a CDF (like Figure 8). However, we also define a numeric summary, CFGINSULATION. In Figure 9, we show the CFGINSULATION values for every benchmark and every CFI policy generator we consider.

The CFGINSULATION of NumCFI is approximately 5 times greater than TypeCFI considering all call sites together. Considering the benchmarks separately, in six of them NumCFI has a higher CFGINSULATION than TypeCFI, while for seven benchmarks TypeCFI has a higher CFGINSULATION. The latter seven are the benchmarks with the lowest amount of basic blocks and, hence, a distance-based CFI policy is less effective for them. For more complex applications, like the other six benchmarks, NumCFI shows a better performance than TypeCFI.

Moreover, Num+TypeCFI has a better CFGINSULATION than TypeCFI in every benchmark. Considering all call sites together, Num+TypeCFI has a CFGINSULATION which is approximately 29 times greater than TypeCFI. For eight benchmarks the improvement is at least tenfold.

Other metrics. We also compare NumCFI and Num+TypeCFI using other the existing metrics we select in Section V: fAIR [55], fAIA [18], iCTR [34] and QS [6]. We report the value of these metrics in Table II. We also report, in parentheses, the rank of every policy generator according to each metric (1 marks the best and 7 marks the worst).

We can make a number of observations from Table II. First, according to all metrics we examine (both our metrics and existing metrics), Num+TypeCFI outperforms both NumCFI and TypeCFI, always ranking second after the baseline. Second, all CFI policy generators have very high fAIR (above 99.9%), proving the point that AIR is not an effective metric to evaluate

the security guarantees of CFI policies. Third, other metrics, like fAIA, iCTR, and QS, show a more significant variation between the policy generators we evaluate. However, they have a disadvantage. They only consider the number of targets that are reachable; yet, they neglect to take into consideration the usefulness of the blocks for an attacker. BLOCKINSULATION and CFGINSULATION overcome this limitation, taking into account the usefulness of each basic block to the adversary.

VIII. L+TCFI

In the previous Section, we evaluate the security of a number of CFI policy generators, including NumCFI and Num+TypeCFI. This Section shows that the generated policies can be implemented in an efficient way, with a low run-time overhead.

To do so, we design L+TCFI, a generic CFI enforcement mechanism. In L+TCFI, every basic block b has two properties, a label l_b and a tag t_b. The label is determined by a label-based CFI policy generator like TypeCFI, while the tag is determined by a distance-based policy generator like NumCFI. The mechanism is designed to allow control flow transfers between a block b and a block c if and only if both of these conditions are met:

$$t_c \geq t_b - 1 \tag{1}$$
$$l_c = l_b \tag{2}$$

Ensuring the enforcement of these conditions requires two components: 1) a way to encode the metadata (tags and labels) in the program itself, and 2) a run-time component that decides whether indirect jumps are allowed depending on the encoded information. We discuss both of them in the following.

A. Metadata Encoding

A common strategy for CFI metadata encoding is to embed it in the executable code itself. As an example, RAP [38] inserts the metadata immediately before the beginning of every function in the program. This way, when the run-time checker needs to decide whether a jump to a pointer should be allowed, it can simply read a fixed number of bytes before the pointer and retrieve the metadata. We use this approach as well.

However, embedding metadata in the executable code must be done carefully, in order to not introduce any incompatibility or vulnerability in the application. We do this by embedding our metadata inside of *CFI marks*. Our CFI marks are interpreted by the CPU as a `nop` instruction, which do not produce any result (the name stands for "no operation"). The advantage of embedding data in `nop` instructions is that, unlike raw data, the processor can execute them without changing its state, so they can be easily inserted into the code during the build process.

On x86, `nop` instructions can have different lengths; we choose the 9-byte variant because of its convenience for our purposes. This longer variant of the `nop` instruction is achieved by encoding information on various operands (register and immediate), which are then ignored by the processor. For our purposes, we can consider the leftmost four bytes of the instruction fixed (bytes 8 to 5 in Figure 10). The following byte (4) has multiple legal values which do not influence the behavior of the processor; we choose value `0x9a`, since it is different from the value in Intel's recommended 9-byte `nop` instruction and thus very unlikely to occur in any regular code. The rightmost four bytes of the instruction (bytes 3 to 0 in Figure 10) can be set to any arbitrary value; we decide to encode the label in bytes 3 to 1, and the tag in byte 0. This layout is advantageous for the run-time checker, as Section VIII-B explains; it allows us to encode approximately 16 million labels and 256 tags, which is sufficient in our testing. We encode tag ∞ as 255, any tag $\geq$ 254 as 254, and any other tag as itself.

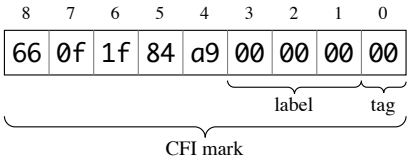

Fig. 10. A CFI mark which embeds metadata in a `nop` instruction.

We insert the CFI marks in the program by 1) instructing the compiler to create an ELF section for every function, and 2) using a custom linker script to insert these instructions between them in the final binary. We insert the marks in assembly files by rewriting them on the fly and adding the required instruction before every function definition.

B. Run-time Checker

The run-time checker has the goal of examining every indirect control flow transfer and decide whether it is allowed or not according to the metadata. We instrument every indirect control flow transfer by developing a custom pass for the Clang C/C++ compiler. Our compiler pass, which was developed for LLVM 11.0.1 and consists of approximately 60 lines of C++ code, finds all indirect function calls and instruments them to check the target address before it is used. Our proof-of-concept implementation does not embed checks in assembly code, which is only a tiny portion of the application code.

The instrumentation code checks that the target address is preceded by a valid CFI mark and that the tags and labels are correct (satisfying Equations (1) and (2) respectively). This can be done with just two comparisons: a single 64-bit equality test can check the presence of a CFI mark and that the labels match (2), while a 8-bit comparison can check whether the target label is greater than the threshold (1).

C. Security Considerations

The goal of L+TCFI is to allow an indirect call if and only if Equations (1) and (2) hold. Our run-time checker (Section VIII-B) is designed to check for CFI marks, which contain the label and tag of a function, before the indirect control flow transfer succeeds. Since we assume W$\oplus$X to be in place (Assumption A3), the adversary is unable to insert counterfeit CFI marks into the application. The adversary could, however, leverage data which accidentally matches the format of a CFI mark and is included in the code of the application. To investigate this possibility, we scanned for the

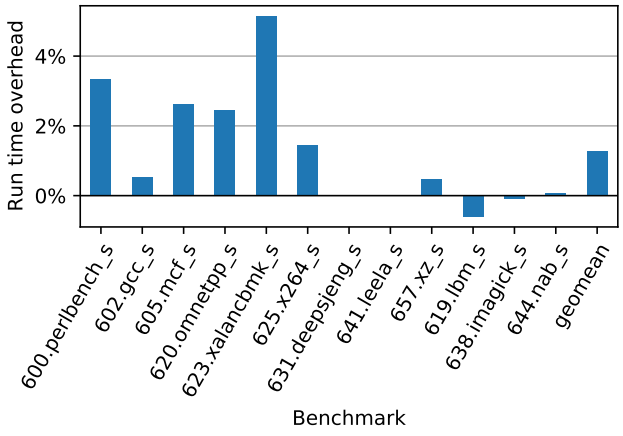

Fig. 11. Run-time overhead of SPEC CPU2017 benchmarks when using L+TCFI.

prefix of our CFI marks (the binary string `660f1f84a9`) all the baseline binaries used in our performance evaluations, as well as all binaries in the directories `/bin`, `/sbin`, and `/lib/x86_64-linux-gnu` on our test system. We did not find any match. We can then assume that accidental matches are very unlikely and, hence, the attacker cannot trick the run-time checker into calling an unintended target.

D. Handling Dynamic Libraries

Binaries are often distributed independently of the dynamic libraries they require to work. As a result, it can be impractical to apply the CFI marks to all the libraries as well as the main binary. This can be addressed by using *trampolines* that intercept indirect function calls between different libraries. Each trampoline has a CFI mark that contains the expected distance of the target function in a different library. As a result, even if the dynamic library is independently updated, the CFI marks on the trampolines remain the same and functionality is maintained.

E. Performance Evaluation

After describing L+TCFI, we analyze its performance using the run time overhead of the benchmarks we selected from SPEC CPU2017 benchmarks, as well as measuring the reduction of available throughput of an nginx instance.

Benchmarks from SPEC CPU2017. First, we analyze the performance of SPEC CPU2017 benchmarks when protected by L+TCFI compared to an unprotected baseline (Figure 11). We run every benchmark three times on the same machine mentioned in Section VII; we report the median of the three values, as recommended by SPEC. The geometric mean of the overheads is 1.27%. Only two benchmarks have an overhead higher than 3%: *600.perlbench_s* (3.33%) and *623.xalancbmk_s* (5.16%). Unsurprisingly, these benchmarks have a higher proportion of indirect function calls compared to other benchmarks.

Nginx Throughput. In addition to SPEC, we also test the effect of L+TCFI on the throughput of an nginx instance. We configure nginx to only use one worker thread, then we

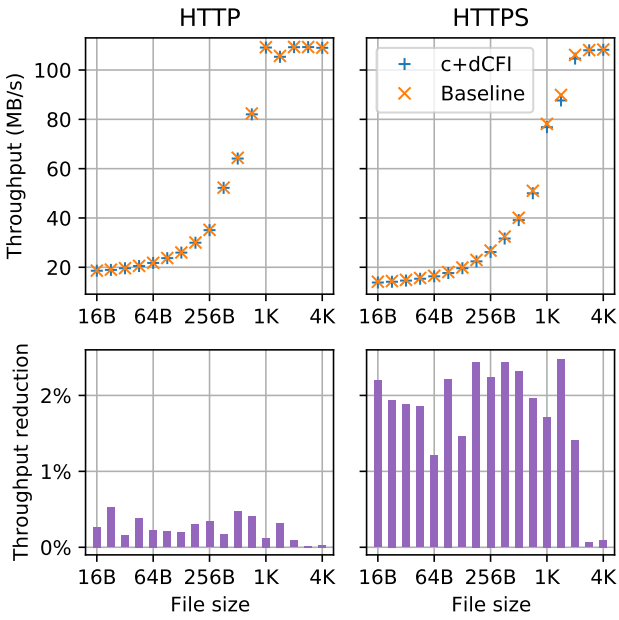

Fig. 12. Throughput and throughput reduction in nginx when using L+TCFI.

run the tool *wrk* [20] on a different machine to determine the connection throughput of a build of nginx protected by L+TCFI compared to an unprotected baseline (Figure 12). The client machine has an Intel Xeon CPU E5-2630 processor and uses its 16 threads to maintain 1024 simultaneous connections. The two machines are connected through a Gigabit Ethernet switch. In our tests we use a number of randomly-generated files having size between 16 bytes and 1 MB, and we access those files through unencrypted HTTP and HTTPS. Our tests show that any file of at least 2 KB is sufficient to saturate the Gigabit Ethernet connection while using HTTP, and any file of at least 3 KB saturates it while using HTTPS; in both cases, there is no measurable overhead above these marks. For that reason, we do not show files bigger than 4 KB in the figure. For smaller file sizes we can measure an overhead: considering only the files of size 1 KB or smaller, the geometric mean of the throughput reduction is 0.29% for HTTP and 1.99% for HTTPS.

Both our tests show performance reductions in the order of 1% to 2%, which attests that L+TCFI can be deployed in practice.

IX. Related Work

In this section we give an overview of works in the fields of CFI policies, benchmarks, as well as attacks on CFI implementations.

A. CFI Schemes

A common design aspect of CFI policies is to assign equivalence classes to every indirect caller and callee, and check that the label of the caller matches that of the callee. The first CFI policy, as proposed by Abadi et al. [2], uses CFGs generated by static analysis to derive labels for valid control-flow transfers between callers and callees, then enforces their match at run

time using inserted checks. Later, Zhang et al. [55] extend this idea to binaries using binary instrumentation. Similarly, Zhang et al. [54] propose a randomized "Springboard section" to encode a CFI policy implicitly by knowing the correct entry for the indirect jump in this springboard. This technique is also used by Tice et al. [51] in combination with a vtable protection to create a fine-grained, forward-edge CFI compiler pass for GCC and LLVM. Lockdown [39] uses dynamic binary instrumentation to inject CFI checks dynamically at run time. For backward-edge protection, Lockdown uses a shadow stack that is also guarded by dynamic checks. However, this flexibility incurs a higher performance overhead.

Another promising approach is enforcing type-based policies to restrict control-flow transfers. For example, TypeArmor [53] leverages binary-analysis techniques to infer the parameter count of a function, to restrict call targets to functions with less or equal amount of parameters than prepared by the caller. τCFI [33] extends this approach by also taking the parameter types into consideration and leverages a points-to analysis for the return instruction to protect the backward edge. MARX [36], as well as VCI [15], augment CFI mechanisms with efficient vtable protection by leveraging reconstructed class hierarchies to reduce the overapproximation of, e.g., type-based CFI policies [33]. Type-based CFI policies often imply a relatively low performance overhead, hence, the clang compiler frontend of LLVM also features a type-based CFI policy [30] that checks a variety of dynamic types. All of these label-based CFI implementations do not consider the distance of blocks to a system call, which we introduce with NumCFI, that prevents the attacker from taking shortcuts from the vulnerability to a system call.

A different line of research investigates *context-sensitive* CFI schemes, which consider some form of context to decide whether an indirect call should be allowed. PathArmor [52] compares the latest 16 taken branches against a statically generated list whenever the application calls sensitive system calls. πCFI [35] dynamically constructs a CFG at run time, in order to restrict the legal targets of return instructions, but it is similar to context-insensitive CFI for forward edges. PittyPat [14] intercepts security-sensitive system calls and validates the control flow of the program based on online points-to analysis of a subset of control-relevant data. μCFI [23] extends this analysis to include more constraint data and further refine the sets of allowed targets from any indirect call. OS-CFI [27] focuses on reducing the size of the biggest equivalence class by leveraging information about the origin of the code-pointer used by the indirect call.

B. CFI Benchmarks

In order to compare the security of CFI policies, it is crucial to quantify and compare how restrictive they are. A number of metrics have been proposed for this purpose. The best-known metric is Average Indirect-target Reduction (AIR) [55], which is defined as the average reduction of allowed targets across every CFG node (the higher the better). However, AIR is not a good metric to compare different policies [51], as most CFI papers that rely on AIR report similar values greater than 99% [6]. Other metrics also have been proposed, e.g., QuantitiveSecurity [6], which is based on the number and size of equivalence classes, AIA [18], which measures the average number of allowed indirect targets, or Calltarget Reduction (CTR) [34], which measures the absolute number of remaining call targets after applying a CFI defense. However, all of these metrics have a common pitfall: they consider every basic block equivalent to each other. The goal of an attacker in most cases is to leak the content of some memory, exfiltrate some files, or install malware on the victim machine. The first goal is trivially possible in the common CFI threat model which includes arbitrary data read capabilities. The second and the third goal require accessing system resources, for which the attacker *needs* to use system calls. While other metrics do not consider whether an edge is useful to allow the attacker to reach a system call, CFInsight is the only CFI evaluation framework that considers this goal rather than merely the count of reachable blocks.

C. Attacks on CFI

Although CFI can be a strong defense even in practical scenarios, bypasses are possible, and can also be found in practice. While Göktas et al. [21], as well as Davi et al. [13], demonstrated that coarse-grained CFI can be bypassed with new types of ROP gadgets due to the low number of labels, the first work presenting a bypass for more secure fine-grained CFI defenses was Control-Flow Bending [7]. It shows that a single `printf` call can lead to Turing-complete computation by abusing the right format specifiers, allowing the attacker to overwrite the return address of `printf`, even in the presence of a fully-precise static CFI implementation. Control Jujutsu [16], instead, exploits insecure programming patterns and the impreciseness of static analysis approaches to find legal but unintended control flows, which can be leveraged for attacks. Most of this impreciseness is caused by the central element needed for CFG creation: the complete points-to analysis for pointers, which is undecidable [42], [22], and, hence, hard to achieve in practice. Another work uses specifics of C++, namely vtable pointers, to chain virtual function calls through existing call sites, allowing the attack to be resistant against even fine-grained CFI enforcement [44]. Recently, multiple CFI bypasses switched to *data-only attacks*, in which the adversary corrupts only non-control data. These attacks can inherently bypass any kind of static CFI, as they chain only legitimate control-flow paths. While there are already solutions that automatically generate payloads [26], [40], they are still limited in target executable size due to heavy use of static analysis.

X. Conclusion

In this paper, we present CFInsight, a novel framework to evaluate the security guarantees of CFI policies. With our novel metrics BlockInsulation and CFGInsulation we measure the usefulness of any basic block to constructing a code-reuse attack targeting a system call instruction. We introduce NumCFI, a novel CFI policy generator based on the distance between each basic block and the closest system call instruction. We use CFInsight to analyze seven CFI policy generators, including NumCFI, using five different metrics, including CFGInsulation. Lastly, we describe L+TCFI, a fast implementation of NumCFI combined with a type-based policy, with a performance overhead of just 1.27% on benchmarks from the SPEC CPU2017 suite.

ACKNOWLEDGMENTS

This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297, by the European Space Operations Centre with the Networking/Partnering Initiative, by Huawei within the OpenS3 Lab, by the German Federal Ministry of Education and Research and the Hessian State Ministry for Higher Education, Research and the Arts within ATHENE, and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 952697).

REFERENCES

[1] "nginx," http://nginx.org.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "CFI: Principles, implementations, and applications," in *Proc. ACM Conference and Computer and Communications Security*, 2005.

[3] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan, "HAFIX: Hardware-Assisted Flow Integrity Extension," in *52nd Design Automation Conference*, 2015.

[4] E. Bendersky, "pyelftools," https://github.com/eliben/pyelftools, Jul 2020.

[5] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *2019IEEE Symposium on Security and Privacy*, 2019.

[6] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Comput. Surv.*, vol. 50, no. 1, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3054924

[7] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium*, 2015.

[8] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[10] W. Chen, "Here's that FBI Firefox exploit for you (cve-2013-1690)," https://community.rapid7.com/community/metasploit/blog/2013/08/07/heres-that-fbi-firefox-exploit-for-you-cve-2013-1690, 2013.

[11] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *36th IEEE Symposium on Security and Privacy*, 2015.

[12] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Design Automation Conference*, 2014.

[13] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium*, 2014.

[14] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *26th USENIX Security Symposium*, 2017.

[15] M. Elsabagh, D. Fleck, and A. Stavrou, "Strict virtual call integrity checking for C++ binaries," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[16] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[17] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi, "On the effectiveness of type-based control flow integrity," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[18] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *2016 IEEE European Symposium on Security and Privacy*, 2016.

[19] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *1st IEEE European Symposium on Security and Privacy*, 2016.

[20] W. Glozer, "wrk - a http benchmarking tool," https://github.com/wg/wrk, Apr 2019.

[21] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*, 2014.

[22] S. Horwitz, "Precise flow-insensitive may-alias analysis is np-hard," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 1, 1997.

[23] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[24] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy*, 2016.

[25] Intel Corporation, "Control-flow enforcement technology preview." https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2017.

[26] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[27] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th USENIX Security Symposium*, 2019.

[28] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[29] A. Limited, "Arm® a64 instruction set architecture: Future architecture technologies in the a architecture profile," https://developer.arm.com/docs/ddi0602/f/base-instructions-alphabetic-order/bti-branch-target-identification, 2020.

[30] LLVM, "Clang documentation, control-flow integrity," http://clang.llvm.org/docs/ControlFlowIntegrity.html, Jul 2020.

[31] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, 1976.

[32] Microsoft, "Control flow guard for clang/llvm and rust," https://msrc-blog.microsoft.com/2020/08/17/control-flow-guard-for-clang-llvm-and-rust/, 2020.

[33] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert, "τCFI: Type-assisted control flow integrity for x86-64 binaries," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2018.

[34] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert, "Analyzing control flow integrity with LLVM-CFI," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, p. 584–597. [Online]. Available: https://doi.org/10.1145/3359789.3359806

[35] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[36] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, "Marx: Uncovering class hierarchies in C++ programs." in *Symposium on Network and Distributed System Security*, 2017.

[37] PaX Team, "Pax address space layout randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt.

[38] PaX Team, "RAP: RIP ROP," 2015.

[39] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2015.

[40] J. Pewny, P. Koppe, and T. Holz, "Steroids for doped applications:

A compiler for automated data-oriented programming," in *2019 IEEE European Symposium on Security and Privacy*, 2019.

[41] Qualcomm Technologies Inc., "Pointer authentication on armv8.3," https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017.

[42] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, 1994.

[43] A. Rimsa, "Cfggrind," https://github.com/rimsa/CFGgrind, Jul 2020.

[44] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[45] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[46] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[47] H. Sidhpurwala, "Hardening ELF binaries using Relocation Read-Only (RELRO)," Red Hat Blog, https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro, 2019.

[48] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE Symposium on Security and Privacy*, 2013.

[49] Solar Designer, "Getting around non-executable stack (and fix)," https://seclists.org/bugtraq/1997/Aug/63, 1997.

[50] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-Assisted Data-Flow Isolation," in *2016 IEEE Symposium on Security and Privacy*, 2016.

[51] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *23rd USENIX Security Symposium*, 2014.

[52] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[53] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *2016 IEEE Symposium on Security and Privacy*, 2016.

[54] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *2013 IEEE Symposium on Security and Privacy*, 2013.

[55] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *22nd USENIX Security Symposium*, 2013.