USAGE OF DYNAMIC ANALYSIS TO STRENGTHEN CONTROL-FLOW ANALYSIS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Priyam Biswas

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

To Sagar, my partner in crime

ACKNOWLEDGMENTS

First and foremost, I am thankful to Dr. Mathias Payer, my major advisor and mentor for giving me the opportunity to conduct research under his guidance. His cheerful energy and motivational power encouraged me every day to become a better researcher. He is one of the smartest persons I know and “System Security” became fun because of him. Supervising someone very sentimental like me may not have been a good experience for him, yet he guided me with patience and care. I hope I would be able to follow his footsteps someday. I an forever grateful to him for being a constant source of inspiration.

I would like to thank my co-advisor, Dr. Christina Garman for introducing me the world of Cryptography and giving me the freedom to explore research ideas. Her close monitoring and thoughtful insights helped me to refine my research projects.

I am also thankful to Dr. Sonia Fahmy, Dr. Xiangyu Zhang and Dr.Aniket Kate, for serving in my dissertation committee, and providing me valuable guidance.

I am grateful to my colleague, Yuseok Jeon, for always being there for me and supporting me from research to life hacks. I would also like to thank all the HexHive group members, Abe Clements, Adrian Herrera, Ahmad Hazimeh, Ahmed Hussein, Alessandro Di Federico, Andrés Sanchez, Atri Bhattacharyya, Antony Vennard, Bader AlBassam, Daniele Antoniolli, Derrick McKee, Hui Peng, Jean-Michel Crepel, Jelena Jankovic, Kyriakos Ispoglou, Naif Almakhdhub, Nathan Burow, Nicolas Badoux, Prashast Srivastava, Scott Carr, Sushant Dinesh, Uros Tesic, and Zhiyuan Jiang for their continous support and precious feedback.

I would like to thank Purdue BARC group members, Arushi Arora, Alex Seto, Devansh Panirwala, Varun Shah, and Yongming Fan for their collaboration and words of encouragement.

I am thankful to my friends Abdullah Al Mamun, Bushra Ferdousi, Marufa Khandaker Joyeeta, and S M Ferdous for always cooking biriyani for me and for their generous support throughout the journey.

This journey would not have been possible without the continous support and motivation from my family. I am grateful to my late mother, Gouri Biswas for always fighting for her daughters' education. I am thankful to my father, Tusher Kanti Biswas, my sister, Dr. Sumana Biswas, and my brother-in-law, Rupam Sarkar for believing in me and their never ending encouragement. Thanks to my nephew, Rick for being our source of happiness. I would also like to extend my gratitude to my father-in-law, Promode Ranjan Chowdhury and mother-in-law, Ratna Chowdhury for their emotional support.

Finally, I am thankful to my husband, Sagar Chowdhury, for making my dreams his own, sacrifing his career to support mine and being always a '+1' for me.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ABI	Application Binary Interface
ASLR	Address Space Layout Randomization
CDF	Cumulative Distribution Function
CFG	Control-Flow Graph
CFH	Control-Flow Hijacking
CFI	Control-Flow Integrity
COP	Call Oriented Programming
CVE	Common Vulnerabilities and Exposure
DEP	Data Execution Prevention
IR	Intermediate Representation
JIT	Just-In Time
LTO	Link Time Optimization
ROP	Return Oriented Programming
VCS	Variadic Call Stack
VCSD	Variadic Call Site Descriptor
VLM	Variadic List Map
VM	Virtual Machine

ABSTRACT

Biswas, Priyam Ph.D., Purdue University, December 2020. Usage of Dynamic Analysis to Strengthen Control-Flow Analysis. Major Professor: Mathias J. Payer.

System programming languages such as C and C++ are ubiquitously used for systems software such as browsers and servers due to their flexibility and high performance. However, this flexibility comes with a price of lack of memory and type safety.

Control-Flow Hijacking (CFH), by taking advantage of the inherent lack of memory and type safety, has become one of the most common attack vectors against C/C++ programs. In such attacks, an attacker attempts to divert the normal control flow of the program to an attacker-controlled location. The most prominent defense against these kind of attacks is Control-Flow Integrity (CFI), which restricts the attack surface by limiting the set of possible targets for each indirect control-flow transfer. However, current analyses for the CFI target sets are highly conservative. Due to the ambiguity and imprecision in the analyses, CFI restricts adversaries to an over-approximation of the possible targets of individual indirect call sites. State-of-the-art CFI approaches fail to protect against special attack classes such as overwriting variadic function arguments. Furthermore, mitigation of control-flow attacks are not explored to its full potential in the context of language boundaries in current literature. Hence, we need effective solution to improve the precision of the CFI approaches as well as strong protection mechanisms against commonly abused corner cases.

We leverage the effectiveness of dynamic analysis in deriving a new approach to efficiently mitigate control-flow hijacking attacks. We present **Ancile**, a novel mechanism to improve the precision of the CFI mechanism by debloating any extraneous targets from the indirect control-flow transfers. We replaced the traditional static analysis approach for target discovery with seed demonstrated fuzzing. We have evaluated the effectiveness of our proposed mechanism with standard SPEC CPU benchmarks and other popular C and C++ applications.

To ensure complete security of C and C++ programs, we need to shield commonly exploited corners of C/C++ such as variadic functions. We performed extensive case studies to show the prevalence of such functions and their exploits. We also developed a sanitizer, **HexVASAN**, to effectively type-check and prevent any attack via variadic functions. CFH attacks, by abusing the difference of managed languages and their underlying system languages, are very frequent in client and server side programs. In order to safe-guard the control-flows in language boundaries, we propose a new mechanism, **FitJit**, to enforce type integrity. Finally, to understand the effectiveness of the dynamic analysis, we present **Artemis**, a comprehensive study of binary analysis on real world applications.

1 INTRODUCTION

1.1 Motivation

C and C++ are popular systems programming languages. This is mainly due to their low overhead abstractions and high degree of control left to the developer. However, these languages guarantee neither type nor memory safety, and bugs may lead to memory corruption. Memory corruption attacks allow adversaries to take control of vulnerable applications or to extract sensitive information.

Modern operating systems and compilers implement several defense mechanisms to combat memory corruption attacks. The most prominent defenses are Address Space Layout Randomization (ASLR) [1], stack canaries [2], and Data Execution Prevention (DEP) [3]. While these defenses raise the bar against exploitation, sophisticated attacks are still feasible. In fact, even a combination of these defenses can be circumvented through information leakage and code-reuse attacks. For example, an attacker can manipulate the control-flow of a program by carefully choosing gadgets within the program; e.g., Call Oriented Programming (COP) [4], Return Oriented Programming (ROP) [5].

Control-Flow Integrity (CFI) [6] is a defense mechanism that prevents control-flow hijacking attacks by validating each indirect control flow transfer based on a precomputed Control-Flow Graph (CFG). While CFI allows the adversary to corrupt non-control data, it will terminate the process whenever the control-flow deviates from the predetermined CFG. The strength of any CFI scheme hinges on its ability to statically create a precise CFG for indirect control-flow edges (e.g., calls through function pointers in C or virtual calls in C++). Due to the dependency on static analysis, traditional CFI approaches cannot resolve aliasing problem and hence, restrict adversaries to an over-approximation of the possible targets of individual indirect call sites. Additionally, traditional CFI approaches fail to provide security against CFH attacks via variadic functions and language boundaries. Therefore, we need effective solutions to shield against all the possible CFH attacks.

1.2 Thesis Statement

This report explores compiler based defense mechanisms to secure applications written in C and C++ as well as inspects the applications of dynamic analysis. Hence, the thesis statement is:

State-of-the-art CFI approaches are over-approximate due to the static nature of the analyses and leave several areas unprotected such as variadic functions and code pointers. We strengthen CFI along these two unprotected dimensions by providing tighter enforcement mechanisms using dynamic analysis and then analyze its applications on real-world programs.

1.3 Contribution

The goal of the thesis report is to secure systems software against CFH-like attack vectors. We present three different mechanisms to effectively mitigate control-flow hijacking attacks by applying dynamic analysis. Our CFI based mechanism **Ancile** is under review for ACM CODASPY 2021, our work on defense against variadic function exploits, **HexVASAN**, was published in USENIX Security 2017, and we are currently working on the prototype of **FitJit** and **Artemis** with an aim to submit them to peer reviewed conferences.

- Ancile
  - We design a mechanism that reduces a program to the minimal amount of required code for a given functionality. We remove the unnecessary code as well as specialize CFI by creating strict target sets to solve over-approximation problem.

  - Our analysis successfully infers code targets based on the user-provided functionality.
  - By re-purposing the efficient LLVM-CFI from a per-equivalence class mechanism to a per-callsite mechanism, we achieve the same performance while significantly increasing the security guarantees through a finer-grained policy.

- HexVASAN
  - By utilizing dynamic call type information, we enforce a tighter bound on variadic function parameters passed on the stack, protecting against type errors and stack overflows/underflows.
  - We have conducted an extensive case study on large programs to show the prevalence of direct and indirect calls to variadic functions.
  - We present several exploit case studies and CFI bypasses using variadic functions.

- Artemis
  - We present a systematic study of cryptographic function identification approaches.
  - We create a standardized suite of performance metrics and benchmarks to evaluate the effectiveness of current detection mechanisms and analyze existing tools based on this suite.
  - Based off of this analysis, we discuss the research gaps in this domain and propose directions for future work.
  - We present a comprehensive framework to understand the scalability and impact of dynamic analysis in detection mechanisms.

- Future Work. In addition, and as an extension to Ancile and HexVASAN, we propose **FitJit** as future work, to enforce type integrity and control-flow integrity to defend against CFH attacks in the context of language boundaries.

2 ANCILE

Modern software (both programs and libraries) provides large amounts of functionality, vastly exceeding what is needed for a single given task. This additional functionality results in an increased attack surface: first, an attacker can use bugs in the unnecessary functionality to compromise the software, and second, defenses such as control-flow integrity (CFI) rely on conservative analyses that gradually lose precision with growing code size.

Removing unnecessary functionality is challenging as the debloating mechanism must remove as much code as possible, while keeping code required for the program to function. Unfortunately, most software does not come with a formal description of the functionality that it provides, or even a mapping between functionality and code. We therefore require a mechanism that—given a set of representable inputs and configuration parameters—automatically infers the underlying functionality, and discovers all reachable code corresponding to this functionality.

We propose **Ancile**, a code specialization technique that leverages targeted fuzzing to discover the code necessary to perform the functionality required by the user. From this, we remove all unnecessary code and tailor indirect control-flow transfers to the minimum necessary for each location, vastly reducing the attack surface. We evaluate **Ancile** using real-world software known to have a large attack surface, including image libraries and network daemons like nginx. For example, our evaluation shows that **Ancile** can remove up to 93.66% of indirect call transfer targets and up to 78% of functions in libtiff's *tiffcrop* utility, while still maintaining its original functionality.

2.1 Introduction

Similar to the second law of thermodynamics, (software) complexity continuously increases. Given new applications, libraries grow to include additional functionality. Both applications and libraries become more complex based on user demand for additional functionality. The Linux kernel is an important example of this phenomenon: its code base has grown substantially over the last 35 years (from 176K LoC to 27.8M LoC [7,8]). Yet, given a single task, only a small subset of a program (or library) is required to be executed at runtime. This increase in code size can also be seen in network facing applications such as nginx or tcpdump, which deal with, e.g., IPv4, IPv6, or proxy settings, as well as image processing libraries, which face increasingly complex file formats as standards expand to support more features. This feature bloat results in a massive amount of unneeded complexity and an ever-growing attack surface. Ideally, applications would be customized with the minimal set of features required by the user, and only the minimum amount of code inlined from imported libraries.

Software complexity results in a flurry of challenges rooted in security, performance, and compatibility concerns. In our opinion, security is the most pressing of these challenges as security flaws can lead to potentially irreversible losses from adversarial exploitation. While functionality may not be required for a given task, adversaries may still find ways to exercise it, increasing the attack surface of a program [9–11]. Additionally, the precision of popular mitigations such as control-flow integrity (CFI) degrades when more code is introduced. Deployed CFI mechanisms [12] leverage function prototypes to disambiguate the target sets of valid targets. Additional complexity increases the probability that functions with the same signature pollute the same target set.

Removing unnecessary functionality is extremely challenging, as the majority of programs and libraries do not come with a formal description of their functionality. Even worse, there is no clear mapping between functionality (i.e., an exposed API) and the underlying code. Reducing the attack surface and removing unnecessary code requires a mechanism to infer this functionality to code mapping based on an *informal description* of the necessary functionality.

Debloating has been embraced by the security research community to remove unnecessary code at various levels of granularity [13–17]. Removing dead code reduces the number of gadgets and unreachable functionality (which may be buggy). Due to the lack of a formal description of functionality, these approaches all remain conservative and must include potentially unneeded functionality. Unfortunately, past research has shown that debloated code still contains vulnerabilities and sufficient targets for an attacker [18].

Our core idea is to facilitate the help of the user who selects the minimum required functionality (by providing a set of example seeds), thus establishing an informal description of functionalities in a program. While this approach was previously used to reverse engineer and extract functional components [19], we are the first to leverage user help to *specialize complex software.* The user provides a set of inputs that exercise the required functionality and a configuration of the software (as part of the environment). Our approach, Ancile, then specializes the program in three steps. First, Ancile infers the required functionality and code through *targeted fuzzing.* Second, Ancile *removes all unnecessary code* in a compilation pass. Third, Ancile computes *minimal CFI target sets* (based on individual indirect call locations instead of over-approximation on function prototypes) to enforce strong security properties.

Note that we propose fuzzing not primarily as a bug finding tool (although Ancile may discover bugs during focused fuzzing that can be reported to the developer) but as a tool for analyzing exercised code. Coverage-guided greybox fuzzing uses code coverage as a feedback to map code to inputs. We use this insight to discover the exercised functionality and to map the corresponding code to user-selected inputs.

The primary contributions of our approach are below:

- We design a code specialization technique that repurposes fuzzing to reduce a program to the minimal amount of code required for a given functionality. Our technique not only removes unnecessary code, but also specializes control-flow checks by creating a reduced target set.

- We present a comprehensive analysis of Ancile on real-world applications to show the effectiveness of fuzzing as a way to generate precise path information.

2.2 Background

We provide a brief introduction of debloating and CFI to minimize the attack surface of applications. We also describe fuzzing and sanitization as these concepts are integral to our approach.

2.2.1 Attack Surface Debloating

To increase software versatility for different users, its size and complexity has grown dramatically over time, resulting in software bloat. For example, a recent study showed that most applications only use 5% of *libc* [15]. This code bloating comes with the burden of increasing the attack surface. Software debloating is a technique that helps prune the program's attack surface by removing extraneous code. Several approaches have been proposed such as debloating via reinforcement learning [14] or trimming unused methods [20]. However, trimming unused or rarely used features cannot alone prevent Control-Flow Hijacking (CFH). By manipulating the remaining indirect call sites, an attacker can still perform code-reuse attacks.

Code debloating improves security along two dimensions: code-reuse reduction and bug reduction. First, code debloating reduces the amount of available code, making it harder for an attacker to find gadgets for a code-reuse attack. Second, feature based code debloating approaches reduce attack surface by removing potentially reachable buggy functionality, making it harder for the attacker to find an exploitable bug.

Unfortunately, security effectiveness of existing code debloating is inherently limited by *the amount of code that remains.* Any functionality in the program requires code, and even tiny programs [21] provide enough code for full code-reuse attacks. While code debloating may be effective in removing some reachable bugs, it is not effective in stopping code-reuse attacks as any remaining code will be sufficient for such attacks.

Debloating restricts attack surface by removing unneeded code, whereas CFI does so by removing extraneous targets from indirect branches. In a sense, code debloating is comparable to Average Indirect Target Reduction (AIR), a metric to measure effectiveness of early CFI mechanisms. Even coarse-grained CFI mechanisms routinely removed more than 99% of targets, yet remained exploitable. An adversary only needs a single usable target but a defense must prohibit all reachable targets to be effective. Partial target reduction is insufficient to stop an attack. Similarly for debloating, the remaining code may still allow the adversary to carry out the attack.

2.2.2 Control-Flow Integrity

Another prominent mechanism for reducing attack surface is Control-Flow Integrity (CFI), the state-of-the-art policy for preventing code-reuse attacks in C and C++ programs. Its key insight is that to perform a control-flow hijacking attack, attackers must modify the code pointer used for an indirect control-flow transfer (direct control-flow transfers are protected as the target is encoded in read-only code). CFI builds, at compile time, a set of legitimate targets for each indirect and virtual call, and, at runtime, validates that the observed target is in the allowed set. By verifying the target, CFI prevents the use of any corrupted code pointer.

State-of-the-art CFI mechanisms have focused on a conservative static analysis for building the target sets which leads to include more targets than the valid ones. This approach has no false positives, but is prone to false negative as it over-approximates targets. It is also possible to use dynamic analysis to construct the target sets, potentially introducing false positives, but greatly improving the precision of the analysis. Here, we discuss both analysis techniques and their trade-offs, for a more in depth survey of CFI see [22].

Static Analysis-Based CFI

Static analysis-based CFI mechanisms compute the allowed target sets at compile time. The goal of the analysis is to discover the set of functions that the programmer intends to target at a given indirect call site. In compiler terms, the analysis is looking for every reaching definition of the function pointer used at the indirect call site. Implementations of the analysis quickly run into the alias analysis problem, and so have to fall back to more tractable, albeit over-approximate, techniques. Early mechanisms reverted to allowing any address taken function [6] to be targeted at any indirect call site. Subsequent mechanisms improved this to any function with a matching prototype [23]. Recent work has even looked at using a context-sensitive and flow-sensitive analysis to further limit the target sets [24, 25]. While such works increase the precision of the analysis, aliasing prevents achieving full sensitivity.

Dynamic CFI

Unlike the static signature-based approach, Dynamic CFI approaches generate or change the target sets of the control-flow transfers during the execution of the program. Dynamic CFI is generally more precise than static CFI as it starts off with a static target sets but then uses runtime information to further constrain the target sets.

Several works have leveraged the support of hardware to restrict the target sets during runtime. πCFI [26] begins with an empty control-flow graph and activates control transfers as required by specific inputs. However, this approach does not execute any address deactivation which may degenerate to the full static control-flow graph (CFG). PathArmor [27] takes advantage of hardware support, specifically the 16 Last

Branch Record (LBR) registers to effectively monitor per-thread control-flow transfers. It limits the verification process to only security critical functions, and verifies the path to these critical functions by using a path cache. PittyPat [28] improves on this by collecting runtime traces via Intel PT, and verifies them in a separate process, halting execution at system calls to synchronize with the verification process. While it is precise (assuming the entire execution is traced), PittyPat also consumes significant additional resources, e.g., another core for the verification process. μCFI [29] improves PittyPat by recording full execution context using Intel PT, and observing unique code target for each invocation of an indirect control-flow transfer. Similar to PittyPat, it relies on a separate monitoring process.

Orthogonally, CFI does not protect against data-only attacks. An attacker that compromises the data of a process can bend execution [9–11] to any allowed functionality and, if a path in the original CFG exists, CFI will allow execution of that path. While CFI limits code execution to legitimate targets under some execution of the program, it does not remove unneeded functionality.

CFI prohibits rogue control flow to unintended locations while code debloating removes unnecessary code. In combination, CFI and code debloating can reduce the exposure of a program but are limited by the remaining code as both approaches are conservative, resulting in an over-approximation of the required functionality.

2.2.3 Fuzzing

Fuzzing [30] is a widely used technique for automatic test case generation. Coverage-based fuzzers such as American Fuzzy Lop (AFL) [31] create a new test case by mutating interesting inputs that trigger new code paths. Their mutation based strategy leads them to test many inputs that cover the same code paths, causing them to explore the possible data-flows of the application as well. Fuzzers operate from a seed input, mutating it in their search for new code-paths while simultaneously exploring data paths as a result of their search.

Ancile requires extensive path coverage, since it is crucial in generating a comprehensive target set for the indirect call-transfers in the desired functionality. Guided fuzzing [32] by modern fuzzing approaches facilitates finding new code paths from an indirect call site. With the knowledge of deeper path information, target discovery has become more efficient.

2.2.4 Sanitization

Sanitization is a dynamic testing technique that effectively detects policy violations at runtime [33]. A sanitizer generally instruments the program during compilation to enforce some security policy. The instrumentation collects metadata about the program execution and continuously checks if the underlying policy is violated.

AddressSanitizer (ASan) [34] employs a specialized memory allocator, and instruments memory accesses at compile time to detect out-of-bounds accesses to heap, stack, and global objects, as well as temporal bugs. ASan is a tripwire-based approach that creates redzones, and checks each memory access to detect memory safety violations. Fuzzing then triggers memory access bugs, allowing ASan to detect them. Apart from ASan, other types of sanitization exist. Memory Sanitizer (MSAN) [35] detects accesses to uninitialized memory by using bit-precise shadow memory at runtime. UndefinedBehaviorSanitizer (UBSan) [36] catches various kinds of undefined behavior during program execution such as null-pointer dereferences.

As Ancile uses fuzzing for functionality inference, we must distinguish between correct functionality and potential bugs. To avoid memory corruption bugs from tainting our allowed functionality, we compile our target program with ASan during the inference phase. Hence, Ancile ensures all the explored targets via fuzzing are indeed valid targets.

2.3 Threat Model

Ancile uses the standard threat model for modern defenses such as CFI and software debloating. We assume that the attacker has the ability to read and write memory arbitrarily. Specifically, we assume that the attacker can modify arbitrary code pointers on the heap and stack to hijack the program's control flow. We also assume that our target system is deployed with the standard software defenses: DEP [37], ASLR [1], and stack canaries [38]. DEP prevents code-injection and forces an attacker to rely on code-reuse attacks. ASLR and stack canaries make attacks harder but do not stop an attack in the given attack model. We include them as they are on by default in modern systems.

Listing 2.1 shows an example of a control-flow hijack attack [39]. In this example, the function `victimFunc` has a buffer, a function pointer and an `int` pointer. By setting `var1` to 128, the attacker causes `ptr` to point to the function pointer on the stack. The dereference of `ptr` at line 8 then causes `var2` to be written to the function pointer. Consequently, an attacker can divert execution to any executable byte at line 9, specified by the value in `var2`. While real-world examples are more complex than this – their spirit is the same. An attacker controlled value dictates a function pointer, virtual table pointer, or return address, thereby hijacking the application's control flow.

Another prominent mechanism for reducing attack surface is Control-Flow Integrity (CFI). It is the state-of-the-art policy for preventing code-reuse attacks in both C and C++ programs. Its key insight is that to perform a control-flow hijacking attack, attackers must modify the code pointer used for an indirect control-flow transfer (direct control-flow transfers are protected as the target is encoded in read-only code). CFI builds a set of legitimate targets for each indirect and virtual call, and validates that the runtime target is in the allowed set. By verifying the target, CFI prevents the use of any corrupted code pointer.

```c
void bar() { }

int victimFunc(int var1, int var2) {
  void (*fnptr)();
  char buffer[128];
  int *ptr = buff + var1;
  fnptr = &bar;
  *ptr= var2;
  fnptr();

  return 0;
}
```

Listing 2.1 Control-flow hijacking example.

To date, CFI mechanisms have focused on a conservative static analysis for building the target sets. This approach has no false positives, but is also fundamentally over-approximate. It is also possible to use dynamic analysis to construct the target sets, potentially introducing false positives, but greatly improving the precision of the analysis. Here, we discuss both analysis techniques and their trade-offs, for a more in depth survey of CFI see [22].

2.4 Challenges and Trade-offs

Code specialization is a technique used to generate more efficient code for a specific purpose from a generic one [40]. The core issue of code specialization is the prediction of effective code-behavior in order to generate precise control-flows. Specializing an application allows us to apply both attack surface reduction techniques at once, by removing code unused by the deployment scenario, *and* restricting targets to exactly the purposefully valid sets. However, automatically specializing code to only support a user specified configuration is challenging. Static analysis quickly degenerates to the aliasing problem [41], and has difficulty determining if a function is required for a particular functionality. Dynamic analysis is an attractive alternative, however, it requires that all valid code and data paths for a particular configuration are explored.

Dynamic analysis has been made practical by recent advances in automatic testing, and in particular coverage-guided fuzzing [31, 32, 42, 43]. Given a minimal set of seeds that cover the desired behavior, fuzzers are capable of quickly and effectively exploring sufficient code and data paths through a program to observe the required indirect control-flow transfers for a given configuration. CFI target sets are then restricted to the observed targets for the desired functionality of the application, e.g., an IPv4 deployment of nginx with no proxy. Note that the dynamic analysis can occur offline, with only traditional CFI set checks, which incur minimal performance overhead, required at run time. *Ancile leverages fuzzing to correlate functionality with code.* Fuzzing's code exploration serves as a mapping process from functionalities to relevant code-regions. The coverage information from fuzzing enables us to effectively specialize software by replacing conservative analysis of valid cases with a more precise analysis of what states are reachable in practice. Using fuzzing as a path exploration technique introduces its own set of challenges: (i) generating a dynamic control-flow graph (CFG) for user-selected functionality, (ii) projection of dynamic CFG in functionality-based debloating, (iii) precision vs soundness in CFI target analysis, and (iv) the risk of introducing false positives and false negatives due to the randomness associated with fuzzing. We now discuss each of these challenges in turn and how we address them.

Challenge i. Generating a dynamic CFG: Given a program with a set of functionalities f_1,f_2, f_3,...,f_n and a user-specified functionality $f_s \subset \{f_1,f_2, f_3,...,f_n\}$, we must discover the code required by that particular functionality, f_s. For example, a user may only require the *tiffcrop* functionality from the image library libtiff. To generate a dynamic CFG for a given functionality, we need to explore all required and valid control-flows exercised by that functionality within the program. Ancile address this by taking as input a set of seeds and configuration demonstrating the required functionality (f_s), and then uses these to fuzz the application in order to retrieve the relevant control flows. We start with an empty CFG and add edges only if their execution is observed in the set of valid executions.

Challenge ii. Projection of dynamically generated CFG in functionality-based debloating: To prune unneeded functionality, we need to map the control-flow information into relevant code. In order to do so, we guide fuzzing by carefully selecting inputs to explore the intended functionality. Similar to Razor [13] and binary control-flow trimming [44], Ancile utilizes test cases to trace execution paths. Ancile also takes advantage of the power of coverage-guided fuzzing to explore deeper code paths pertinent to the desired functionality. To ensure that the fuzzed functionality has covered all possible paths, we evaluate the targeted utility with a different set of testcases. Ancile then removes any functions that have not been triggered during fuzzing.

Challenge iii. Precision vs soundness: Ancile trades theoretical soundness for precision when constructing CFI target sets. State-of-the-art CFI mechanisms have focused on a conservative static analysis for building the CFG, resulting in a conservative over-approximation of indirect control-flow targets. These CFI mechanisms quickly run into the alias analysis problem, and so must fall back to more tractable, albeit over-approximate, techniques. Recent approaches have looked at using context-sensitive and flow-sensitive analyses to further limit the target sets [24,25]. While such works increase the precision of the analysis, aliasing prevents achieving full sensitivity.

It is also possible to use dynamic analysis to construct the target sets, potentially introducing false positives, but greatly improving the precision of the analysis. Several works [26–28] introduce hardware requirements to restrict the target sets during runtime. Both static and dynamic approaches are inherently over-approximative as existing CFI solutions are oblivious to a minimal, user-specified functionality. Static analysis-based approaches leverage only information available during compilation, while dynamic analysis-based approaches use runtime information to further constrain the target sets. Still, existing dynamic mechanisms result in over-approximation in the target set. Ancile extensively fuzzes the desired functionality to infer the required control-flow transfers. Fuzzing's efficiency comes from its fundamental design decision: to embrace randomness and practical results rather than theoretical soundness. Consequently, fuzzing gives no guarantees about covering all possible code or data paths, but covers them well in practice.

Challenge iv. False positives and false negatives: Our goal is to minimize the number of targets for individual CFI checks. Ancile restricts per-location CFI targets by combining per-function removal along with CFI-based target removal. An unintended function included in the target set is a false negative This can happen in two scenarios, (i) a fuzzing campaign performing invalid executions; and (ii) exploring traces outside of the desired functionality. Ancile guarantees valid executions by using Address Sanitizer (ASan) along with fuzzing. Furthermore, by restricting our fuzzing campaigns to only the intended functionality, we guide our fuzzing campaigns, cautiously selecting the input seeds as well as tuning the fuzzing campaign.

A false positive happens if a valid and intended target is not included in the generated set. This may happen due to lack of fuzzing coverage. Ancile starts with the minimum set of seeds that exercise the intended functionalities, giving a lower-bound of targets. Next, fuzzing discovers targets that were not previously included. Moreover, to increase confidence in the discovered target set, we repeat each fuzzing campaign multiple times. We explore the issue of false positives/negatives further in Section 2.7.

2.5 Ancile Design

Based on the user-selected functionality (through provided seeds), Ancile generates specialized binaries. The design of Ancile is motivated by the need for precise control-flow information so that this information can be used to debloat the target program, *reducing its exposed attack surface*. The user *informally specifies* the desired functionality by providing seed inputs that explore that functionality. Ancile operates in three distinct phases, as shown in Figure 2.1. First, Ancile performs targeted fuzzing (using the seeds provided by the user) to infer the CFG and to explore code associated with the required functionality (including error paths). This step infers all of the necessary information for the next two steps. Second, Ancile removes any unnecessary code using a compiler pass, reducing the program's attack surface. Third, Ancile leverages the precise CFG to customize CFI enforcement to the observed CFG. This customization increases the precision of CFI to only observed targets. These observations result in the following requirements:

Desired Functionality. Every application has its own set of features. By desired functionality, we mean one or more features of the application that the user intends to exercise. For example, in tcpdump, the user may only want to exercise the feature that reads `pcap` files.

Seed Selection. The minimum number of inputs required to exercise the desired functionalities is selected. For example, to exercise the feature of reading a `pcap` file, the user only needs to provide a captured `pcap` file.

User Involvement. Ancile requires two sets of input from the user, (i) necessary command line arguments to select the functionality; and (ii) a minimum set of seeds that exercise this functionality. For reading a `pcap` file, the user must provide (i) the `-r` command-line argument, and (ii) a `pcap` file as an input seed.

The key insight of Ancile is the functionality analysis. It is this analysis which allows us to *automatically* specialize an application, simultaneously removing extraneous features and shrinking the attack surface by restricting the set of allowed indirect control-flow transfers. Selection of the required functionality depends on the type of application as well as user requirements. Ancile minimizes the user burden for feature selection. For example, if a user wants to read `pcap` files using tcpdump, she will configure Ancile to execute tcpdump with the command line option `-r`, and a sample `pcap` file as input. Ancile also takes advantage of existing unit test-suites that comes with the application package to exercise functionality.

Ancile uses fuzzing to infer the code covered by an informally-selected functionality. Input seeds are used to exercise the desired functionality. Coverage-based fuzzing excels at finding code paths from a given seed. For each target in our per CFI-location target sets, fuzzing produces an execution that witnesses that specific target. The challenge becomes ensuring that the set of executions used by our functionality analysis fully covers the control and data flows of the desired functionality. We show that fuzzing, in conjunction with a small set of test cases that observe the desired functionality, can be leveraged to generate a precise CFG.

Ancile then utilizes the dynamic CFG constructed in the dynamic CFG generation phase as a mechanism for (i) debloat unnecessary code and (ii) tighten CFI checks to restrict indirect control-flow to a set of targets required by a given user specification. Ancile can achieve the best possible precision with negligible runtime overhead, i.e.,

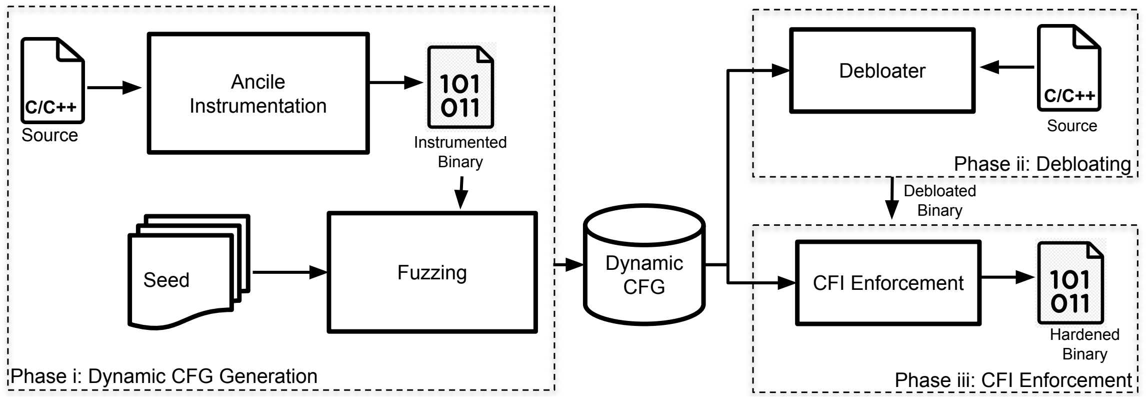


Figure 2.1. Ancile operates in three distinct phases: (i) Dynamic CFG Generation (to record control flow), (ii) Debloating (to remove unnecessary functionality), and (iii) CFI Target Analysis (to tighten indirect control flow checks to the minimal required targets).

set checks inserted at compile time. Therefore, we believe that increased specialization is the way of the future for “prevent-the-exploit” defenses.

2.5.1 Dynamic CFG Generation

Ancile requires the user to select the desired functionality of the program by providing corresponding input. These input seeds can come from, e.g., unit tests, examples, or be custom tailored by the user. For example, the network sniffer *tcpdump* offers a variety of features, from directly capturing network packets to processing recorded traces. A user may want to only process recorded traces of a single protocol. Building off this informal specification, Ancile performs dynamic fuzzing that identifies (i) all the executed functions, and (ii) the targets of indirect function calls. Any function that has not been observed via direct or indirect calls during this phase is considered extraneous and hence, is not included in the CFG. At this point, our analysis is fully context and flow sensitive, as it directly depends on actual executions.

After this analysis, the observed targets are aggregated over each indirect call site. This aggregation results in some over-approximation and a loss of full context and data sensitivity. However, every target we allow is valid for *some* execution trace, which is a significantly stronger guarantee than is provided by static analysis-based CFI [22]. Static analysis-based target sets only guarantee that every target *may* be required by an execution trace. Put another way, our dynamic analysis recovers the *programmer-intended target sets*, rather than an over-approximation thereof.

Ancile recompiles the application with not only the coverage instrumentation for grey box fuzzing, but also to log the targets for direct and indirect control-flow transfers. In particular, we cover forward edges, leaving return edges for more precise solutions such as a shadow stack [45]. When running the fuzzing analysis, we use AddressSanitizer [46] to validate that all observed executions are in fact valid and free of memory errors.

As fuzzing is incomplete, the core risk of this approach is that some required functionality is not discovered and therefore unintentionally removed. Our analysis could potentially introduce false positives (prohibiting valid indirect control-flow transfers). This is in direct opposition to the conservative approach employed by static analysis, which over-approximates and thus weakens security guarantees. In contrast, Ancile only allows the targets for a particular functionality.

The increased security guarantees through this specialization provide a new avenue for the security community to explore. Our evaluation Section 2.7 shows that with the increasing power of automated testing techniques such as fuzzing [31], robust test sets maintained by many projects [47,48], and a wealth of prior work on sanitizers [46] to validate execution traces, Ancile does not cause false positives in practice.

2.5.2 Debloating Mechanism

In automatic code specialization, unneeded code is discarded and the debloated program contains only the required functionality. Given the user's functionality selection, the challenge of debloating comes from mapping functionality to code regions. One possible approach to address this challenge is to learn code regions through valid program executions that exercise the desired functionality. In other words, we require a set of inputs that exercises, at least minimally, all desired functionality.

By taking advantage of the dynamic functionality observation performed in the first phase of our analysis, Ancile discovers all reachable and executable code. This code analysis can be considered a simple marking phase that records all reachable code. Based on the recorded execution traces, Ancile removes all unneeded code. As a second compilation pass, with the marked code from the fuzzing campaigns, we then tailor and remove all unnecessary code on a per function basis. All functions that are unreachable are replaced with a single empty stub. If this stub is reached, the program is terminated with an error message.

2.5.3 CFI Target Analysis

Although, debloating restricts a program's attack surface by removing unneeded code, it is still possible that vulnerabilities remain in non-bloated code. To ensure tighter security in the specialized binary, Ancile removes extraneous targets from indirect control-flow transfers in the remaining code.

The main goal of Ancile's CFI target analysis is to achieve minimal target sets for indirect branches. It does so by only allowing targets that are required for the specified functionality *and actually observed at runtime.* For each target, we ensure that there is at least one dynamic witness, i.e., a valid execution trace that includes the indirect call. Hence, Ancile solves the aliasing problem of static analysis based approaches and increases precision.

Based on the inferred CFG that is tied to the actual execution of the desired behavior, Ancile learns—for each indirect control-flow transfer—the exact set of targets observed during execution. This set is strictly smaller than the set of all functions with the same prototype. Once the target sets are created, we recompile the application to a *specialized* form, which enforces the target sets derived from our functionality analysis.

Since we focus on static CFI enforcement mechanisms, deciding if a target is allowed depends purely on the information known at compile time, regardless of how that information was obtained. For example, if two paths in a program result in two different targets at a location then the most precise static mechanism will always allow both targets (as it cannot distinguish the runtime path without tracking runtime information). In contrast, dynamic enforcement mechanisms can modify the target sets depending on runtime information (e.g., data-flow tracking). Unfortunately, dynamic mechanisms result in additional runtime overhead (e.g., to update the target sets), increased complexity (for ensuring that the target sets remain in sync), and compatibility issues (e.g., the runtime metadata for the CFI mechanism must be protected against an adversary during the updates). For as long as no hardware extension exists for protecting metadata (e.g., to protect attacker-controlled arbitrary writes from the buggy program), realistically deployable CFI mechanisms will remain static.

2.6 Implementation

Ancile is implemented on top of the LLVM compiler framework, version 7.0.0. The LLVM-CFI framework has entered mass deployment [49, 50], and its set checks are highly optimized. Consequently, building on top of LLVM-CFI guarantees that our enforcement scheme is efficient, and ready for wide-spread adoption. As mentioned in the design, the Ancile implementation constitutes three parts: (i) dynamic CFG generation, (ii) debloating and (ii) CFI enforcement, following the description in Section 3.4.

Dynamic CFG Generation This functionality analysis phase is implemented as a combination of an LLVM compiler pass and a runtime library. Our instrumentation takes place right after the clang front-end and modifies the LLVM IR code. Ancile is enabled by specifying our new `fsanitize=Ancile` flag.

C/C++ source files are first passed to the clang front-end. The compiler pass adds instrumentation to log all indirect calls and their targets. At the IR level, Ancile adds a call to the logging function in our runtime library before every indirect call. The logging function takes two arguments: location of the indirect call in the source, as well as the address of the targeted function. Additionally, the pass logs all the address taken functions to facilitate the remapping of the logged target addresses to corresponding functions. The runtime library of Ancile generates a hash map to store target set information per call site. To remove extraneous code, Ancile collects information during profiling about function invocations via direct control-flow transfers. This procedure follows the same mechanism described above for indirect control-flow transfers. Hence, Ancile generates a dynamic CFG accommodating all the observed control flows that reflect the user specified functionality.

The challenge associated with fuzzing is to guarantee that paths taken during fuzzing are valid code and data paths. To address such challenges, we leverage AddressSanitizer (ASan) [34], a widely-used sanitizer that detects memory corruptions (e.g., use-after-free or out-of-bounds access). Only non-crashing executions are recorded. Hence, Ancile ensures all the recorded control-flow transfers are from valid execution traces and generates the dynamic CFG.

Debloating To prune unnecessary code, Ancile utilizes the dynamic CFG to construct the list of observed functions. It then removes any functions that are not in our observed white list, thereby ensuring a custom binary incorporating only the user specified features. It relies on a compiler pass to remove any unintended function.

CFI Mechanism Ancile enforces the strict targets for the indirect calls based on the dynamic CFG. Despite relying on dynamic profiling, Ancile still enforces target sets statically (i.e., relying only on information available at compile time to embed the target sets in the binary). We have customized LLVM-CFI to adopt Ancile's strict target set at each individual indirect control transfer check points. Our target-set sizes are smaller in most cases and equal to the size of the LLVM analysis in the worst case. In contrast to Ancile, vanilla LLVM-CFI relies on static analysis for target generation and thus fails to solve aliasing, resulting in an over-approximate target sets. The main advantage behind adapting LLVM-CFI is that it is highly optimized and incurs only 1% overhead [12]. Our framework for using LLVM-CFI to enforce user-specified target sets will help the research community to advance control-flow hijacking mitigation by serving as an enforcement API for any analysis that generates target sets.

2.7 Evaluation

The evaluation of Ancile is guided by the following research questions:

RQ1. Can fuzzing be used to enable debloating?

RQ2. Can fuzzing be used as a CFI target generator?

RQ3. How can we analyze the correctness?

RQ4. How performant is Ancile (in particular, compared to LLVM-CFI)?

We performed a series of investigations on Ancile to answer the research questions posed above. For our evaluation, we selected commonly attacked diverse software that offers rich opportunities for customization and specialization. We chose two popular, and frequently attacked, image libraries `libtiff` and `libpng`, as well as two network facing applications, `nginx` and `tcpdump` which deal with different proxy settings for our analysis. To show the impact of feature selection, we investigated four different cases for each of the applications. We analyzed vanilla LLVM-CFI and Ancile with the application's standard test-suite (included in the package), as well as two user-selected functionality sets. For the two image libraries, we use the utilities *tiffcrop*, *tiff2pdf* for libtiff and *pngfix*, *timepng* for libpng. We used a set of *tif* and *png* files as input seeds to fuzz the libraries respectively. For tcpdump, we leveraged two sets of command line arguments `-r` and `-ee -vv -nnr` as well as network capture files in the `cap` and `pcap` formats as input seeds. For nginx, we used methods such as GET, POST, and TRACE operations as inputs along with two different configuration settings.

2.7.1 Effectiveness of fuzzing as a debloating tool (RQ1)

With the advancement of efficient coverage-guided mechanisms, fuzzers can be used to observe valid code executions. Ancile learns valid targets yielding from valid execution paths. Ancile utilizes mutational fuzzing via AFL and honggfuzz to explore relevant code paths. To generate complete observed function sets for a desired functionality, it is possible to carefully select input seeds for that particular functionality. For instance, if the user only wants to read `pcap` files via tcpdump, we can provide only `pcap` files as seed. In the case, where the user wants to read both `cap` and `pcap` files, we can then use both type of files as seeds.

In the following sections, we have analyzed fuzzing's effectiveness in debloating and CFI checks. Fuzzing has been mainly used as a bug finding mechanism. To demonstrate its effectiveness as a debloating mechanism, we evaluate code reduction by Ancile on our case studies. Additionally, Ancile improves the security of the debloated binary by pruning gadgets as well as security-sensitive functions. All performance measurement were done on Ubuntu 18.04 LTS system with 32GB memory and Intel Core i7-7700 processor.

Function Debloating Ancile debloats applications by removing all unused functions, i.e., code that was never executed during our functionality inference phase. It generates a white list of functions based on the context of the user-specified functionality and removes functions that were not invoked during execution. Figure 2.2 compares the number of functions before and after debloating is performed across different benchmarks. Additionally, function reduction depends on the specified functionality. Ancile reduces around 60% functions for libtiff standard test-suite that comes with the library, where as for a more specialized scenario, for example in case of *tiffcrop* utility, reduces 78% functions.

Pruning-Security Sensitive Functions The main goal of Ancile is to allow the minimum set of control-flow transfers for the required functionality, thereby minimiz-

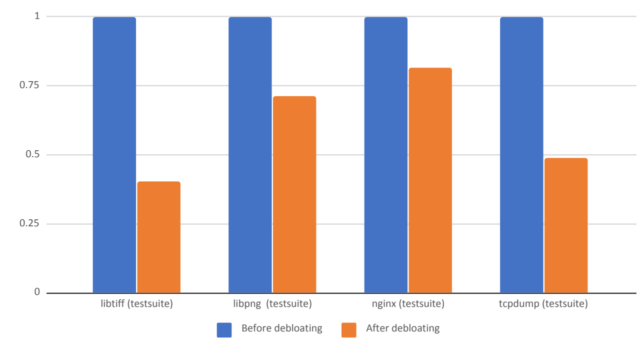


Figure 2.2. Comparison of the number of functions before and after debloating across our benchmarks: libtiff, libpng, tcpdump, and nginx. We used the standard test-suite for each of these applications. Ancile reduces more functions in specialized cases.

ing the available attack surface. Sensitive functions belonging to a target set increase the attack surface. We measure if sensitive functions are reachable from (i) indirect calls i.e., they are in the target sets, (ii) at distance-1 (indirection +1), i.e., if a function in the target set calls a sensitive function, (iii) at distance-2 (indirection +2), i.e., if a function in the target set calls a function that calls a sensitive function, and (iv) similarly at distance-3 (indirection +3). In short, we have observed different level of indirect calls in the evaluated benchmarks. We considered `execve`, `mmap`, `memcpy`, and `system` as the set of sensitive functions in our analysis. The main reason behind selecting such functions as sensitive is that an attacker can modify the arguments of these functions such as `system` to execute unwanted actions and gain control of the system. Since, there were no security sensitive function directly in the target set, we exclude criterion (i) from our analysis.

Table 2.1 shows reachability to sensitive functions from an indirect call site through a sequence of intermediate calls. For instance, in libpng several calls are made to the sensitive function `memcpy`. At indirection+1, indirection+2, and indirection+3 level, there are five, 20, and 17 reachable calls respectively in LLVM-CFI. Ancile restricts these calls to three locations at indirection+1 and in rest of the two cases there are no indirect call sequences to `memcpy`. We have observed another interesting case in nginx, where `execve`, a highly sensitive function, is reachable in indirection+1 in LLVM-CFI, however, Ancile does not allow this call. This call is only made in one rarely-used feature (to hot restart nginx without losing connections when the underlying binary is replaced with a newer version). This demonstrates that focusing on control-flow transfers based on functionality reduces the attack surface when such features are restricted.

Case Study: Gadget Reduction To better understand the significance of Ancile, we performed a case-study on gadget discovery. We focused on two metrics: (i) Jump Oriented Programming (JOP) gadgets, and (ii) unintended indirect-call gadgets. We did not consider ROP gadgets since our framework is aimed for securing forward edges only and CET [51]-like technology will secure backward edges. We built two versions of nginx: one with LLVM-CFI enforcement and the other with Ancile enforcement along all the unit test-suite features. Using a gadget-discovery algorithm and manual analysis, we observed a 54% reduction in JOP gadgets and a 44% reduction of unintended indirect-call gadgets. This case study shows us that Ancile can indeed help in reducing the number of gadgets in an application.

2.7.2 Effectiveness of fuzzing as a CFI tool (RQ2)

To show the effectiveness of fuzzing as a CFI analysis tool, our aim is to establish that fuzzing is effective in producing *drastically* smaller target sets for indirect control-transfers than previous approaches. We found that Ancile can reduce target sets by 93.66% and 97.94% for the *tiffcrop*, *tiff2pdf* utilities from the libtiff image library. Target set reduction reduces the attack surface, increasing the security of our customized binaries. Any additional target which is not intended to be taken during valid program execution potentially increases an attacker's capabilities. We compare Ancile's target set per call site with LLVM-CFI on libtiff-4.0.9, libpng-1.6.35, nginx-1.15.2 and tcpdump-4.9.0, as well as the SPEC CPU2006 benchmark suite.

Table 2.1.
Sensitive function analysis: Number of indirection level to the sensitive functions from functions present in the target sets of LLVM-CFI and Ancile.

Benchmark	Function		ind. +1	ind. + 2	ind. + 3
libpng	`memcpy`	LLVM-CFI	5	20	17
		Ancile	3	0	0
nginx	`execve`	LLVM-CFI	1	0	0
		Ancile	0	0	0
	`memcpy`	LLVM-CFI	1271	2276	2869
		Ancile	167	272	352
	`mmap`	LLVM-CFI	0	2	4
		Ancile	0	1	1
libtiff	`memcpy`	LLVM-CFI	59	95	66
		Ancile	14	14	11
	`mmap`	LLVM-CFI	1	0	0
		Ancile	1	0	0
tcpdump	`memcpy`	LLVM-CFI	156	670	678
		Ancile	34	22	26

To understand the differences in target set generation from different feature selections, we have analyzed the target applications with different user specifications and input seeds. Varying the input seeds for a given specification allows us to examine the effect of path exploration during fuzzing on target set generation.

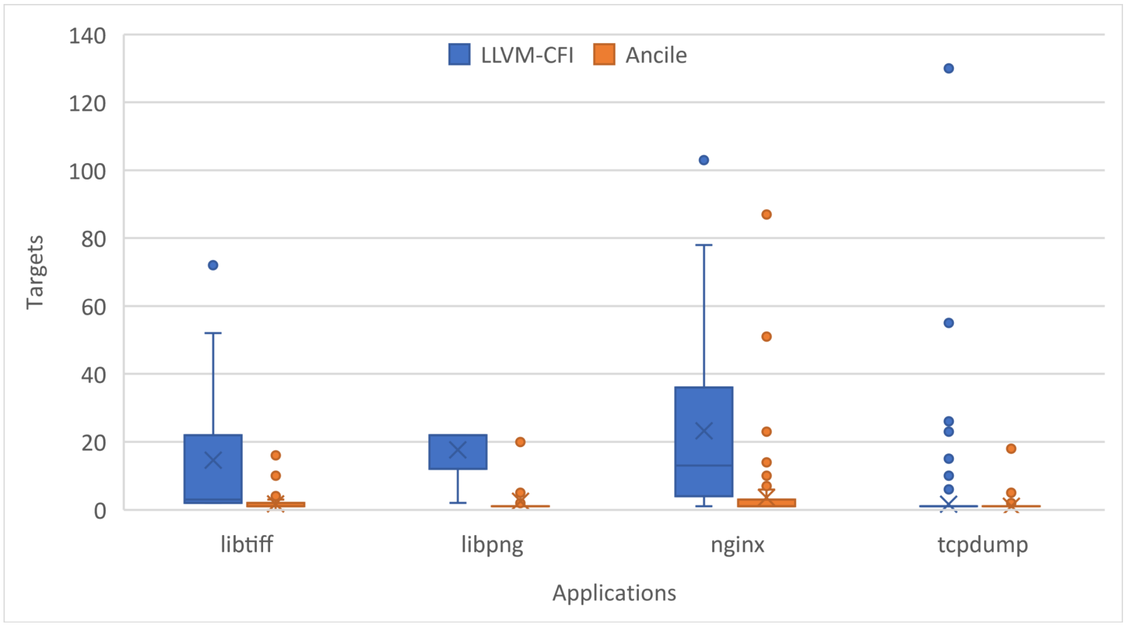


Figure 2.3. Mean and std. deviation of target sets across the four applications in our test-suite for LLVM-CFI and Ancile. LLVM-CFI has more callsite outliers with large target sets than Ancile.

Figure 2.3 shows the mean and standard deviation of target set per call site across the four benchmarks for Ancile and LLVM-CFI. We leverage the application's standard test-suite for Ancile's functionality analysis. In each of the benchmarks libtiff, libpng, nginx and tcpdump, LLVM-CFI has on average 73% more targets than Ancile. Furthermore, LLVM-CFI has outliers of call sites with very large target sets. For example, tcpdump has 48 call sites for which LLVM-CFI reports 130 targets, whereas Ancile observes none to at most two targets. To support our claim in target reduction, Table 2.2 shows the comparison between LLVM-CFI and Ancile for the maximum target set size for each of the benchmarks. This highlights the power of functionality analysis in reducing the attack surface available to attackers.

Figure 2.4 shows the comparison of target-set size per call site between LLVM-CFI and Ancile specializing on different functionalities. In each of the cases, we analyzed

Table 2.2.
Statistics of maximum target size in LLVM-CFI and Ancile for our benchmarks.

Benchmark	Max. target set size	
	LLVM-CFI	Ancile
400.perlbench	354	175
401.bzip2	1	1
429.mcf	-	-
433.milc	2	2
444.namd	40	1
445.gobmk	1642	492
447.dealII	11	2
450.soplex	7	1
458.sjeng	10	6
462.libquantum	-	-
464.h264ref	12	10
470.lbm	-	-
473.astar	1	1
482.sphinx3	5	1
libtiff	78	16 (testsuite)
libpng	48	25 (testsuite)
nginx	103	87 (testsuite)
tcpdump	130	18 (testsuite)

target sets obtained from the unit test-suite as well as target sets obtained from the specialization of certain features as mentioned in Section 2.7. As expected, Ancile reduces the target set sizes for *all* targets, compared to LLVM-CFI. Additionally, fuzzing a particular utility can lead to discovering more targets than the unit test-suite. For instance, for certain indirect control-flow transfers, we observed more targets while fuzzing *tiffcrop* than just running the test-suite.

SPEC CPU2006 In addition to our real-world applications, we also evaluate our prototype on the SPEC CPU2006 benchmark-suite. Working with SPEC CPU2006 enables us to compare with LLVM-CFI. Furthermore, SPEC CPU2006 is the standard performance benchmark, so we included our analysis results for completeness. We used the smaller `test` SPEC benchmark configuration as our functionality specification, and ran the benchmarks once without fuzzing. These target sets were then used to specialize the binaries, and we verified they run with larger `ref` data set, see Section 2.7.4.

Figure 2.5 shows the comparison of Ancile, and LLVM-CFI on two SPEC CPU2006 benchmarks, namely 400.perlbench, and 445.gobmk. We chose to focus on these benchmarks as they have the largest number of indirect call sites. We show the cumulative distribution function (CDF) of target set size per call site. The goal is to have as many call sites as possible and a very short tail, indicating few call sites with many targets, as such call sites are easily exploitable. For example, in case of 400.perlbench 2.5(a), most of the call sites have very few targets, 65% of all call sites have only one target. Similar situations were observed in the 445.gobmk benchmark; where the maximum target set size for LLVM-CFI is 1642, compared to 492 for Ancile. In all of these benchmarks, Ancile has fewer targets than LLVM-CFI as well as the maximum number of targets allowed by any call site is on average 59% smaller. Table 2.2 shows the maximum target set size in LLVM-CFI and Ancile for each of the evaluated benchmarks.

Equivalence Classes Equivalence classes are an important part of static analysis-based CFI. Each class is a group of call sites that are all assigned to the same target set (e.g., based on function prototypes). Ancile does away with the notion of equivalence classes as each call site is independently analyzed, instead of being grouped together as per existing static analysis-based approaches. In other words, Ancile introduces an equivalence class for each indirect call instead of, in its most precise form, for each function pointer type for LLVM-CFI. Having more equivalence classes increases the security of applications [22], as each call site has the minimum target set appropriate for it, not the target set for a class of call sites.

Figure 2.6 shows the equivalence class data for SPEC CPU2006. The ideal scenario is to increase the number of these classes as well as to reduce the size of each class. Ancile breaks large equivalence classes into smaller ones, namely one class per indirect call site, thus restricting the indirect calls to fewer targets. Figure 2.6 shows a comparison between LLVM-CFI and Ancile based on the number of equivalence classes. In the plot, the x-axis corresponds to benchmarks, while the y-axis represents the total number of equivalence classes in each benchmark. Vanilla LLVM-CFI does not compile for five of the benchmarks (403, 453, 456, 471 and 483), hence we did exclude them from the graph. Finally, Ancile generates more equivalence classes than LLVM-CFI, and the classes are strictly smaller, in most cases restricting the call site to single target.

2.7.3 Analyzing the correctness of the specialized binary (RQ3)

To confirm the correctness of Ancile-generated binaries, we performed a series of analyses such as result consistency, assessment of target discovery, correctness of generated input, target set minimality, and statistical analysis.

Consistency One way to establish the confidence of the result is to check for consistency. If two separate fuzzer can generate same set of targets, it can increase our confidence in the specialized binary. We have used two separate fuzzers, AFL and honggfuzz, to generate the dynamic CFG and we achieved similar outcomes.

Target Discovery Using fuzzing for target discovery comes with the challenge of effectiveness in learning targets. To understand this aspect, we plotted the discovery of each unique target against time. Figure 2.7 shows the number of targets discovered over time by the fuzzer for *tcpdump* with the command line option `r` for reading IPv4 and IPv6 captured packets. The `x`-axis plots time in hour and `y`-axis plots the percentage of target discovery. From the figure, it is evident that most of the targets are discovered at the very beginning of the fuzzing procedure and few to no new targets towards later phases of fuzzing. This same observation holds true for all programs we tested. Furthermore, we reran all the fuzzing executions multiple times and target discovery remain identical in all the fuzzing sessions.

This profile of target discovery, with most targets discovered early, increases our confidence that fuzzing is finding all possible targets, and that continuing to fuzz for greater than 24 hours will not find additional targets.

Correctness of Generated Input In order to cross-check that the fuzzer generated executions are valid, we applied several sanitizers (ASan, Ubsan) to check the correctness of fuzzer generated inputs. We also manually ensured that for each of these generated inputs there is an intended control-flow execution.

Minimality Almost all dynamic CFI policies [26] have a fallback strategy and they usually fall back to over-approximated target sets generated statically. Ancile is inherently more aggressive. Although it uses instrumentation similar to LLVM-CFI for its enforcement, it never reduces precision to LLVM-CFI target sets. Ancile considers any call site or target that has not been exercised during profiling phase as invalid or, in other words, not relevant to the intended functionality. This is to ensure that we only employ the desired functionality. Our investigation indicates that this reduction has a *meaningful* impact on the application's security by making sensitive functions harder to access (more levels of indirection are required) from indirect call sites.

Statistical Analysis A potential issue of using fuzzing is that the fuzzer may include superfluous coverage, i.e., the fuzzer discovers functionality that the user does not want included, preferably known as false negative. One way to handle this situation is to tune the length of the fuzzing campaigns. For example, when extracting functionality of reading the captured `pcap` packets using *tcpdump*, it is unlikely that the fuzzer will mutate the input seed enough to discover the code that handles capturing packets. Due to the stochastic nature of fuzzing, it is also possible that Ancile might miss some intended control flows resulting in false positives.

To understand how Ancile performs with respect to false positives and false negatives, we have analyzed it with forty different test cases for each of our case studies. In half of our test cases, we analyzed the specialized binary with the same intended functionality but with different set of inputs. For example, in case of *tiff2pdf* utility, we evaluated it with twenty different *tif* files which we have not used as seed. In similar way, we have used the rest twenty of the test cases to exercise an unintended functionality. Ancile successfully validated all test scenarios for all the investigated applications.

In future work, we will evaluate how a user can select negative functionality they want explicitly excluded. We refer to existing work that focused on similar challenges [19].

2.7.4 Performance Overhead (RQ4)

Performance overhead is crucial in any mechanism, hence we analyzed the performance of Ancile on SPEC CPU2006 benchmark suite and compared it with LLVM-CFI. Table 2.3 presents a comparison of runtime performance of Ancile and LLVM-CFI. Ancile's enforcement mechanism mainly reuses the enforcement part of LLVM-CFI with a tighter target set, and as the table shows, has equivalent runtime performance. As is standard, we report results for three SPEC CPU2006 iterations. Note that we require no additional system resources, such as additional processes, cores, virtual address space, or hardware extensions, unlike other works aimed at increasing the precision of CFI [27,28,52].

Table 2.3.
Performance overhead comparison between LLVM-CFI and Ancile.

Benchmark	Baseline (ms)	LLVM-CFI (ms)	Ancile (ms)
400.perlbench	374	379 (1.33%)	378 (1.07%)
401.bzip2	726	730 (0.55%)	730 (0.55%)
403.gcc	781	-	790 (1.1%)
429.mcf	296	297 (0.34%)	297 (0.34%)
433.milc	1029	1037 (0.78%)	1036 (0.68%)
444.namd	1420	1429 (0.63%)	1430 (0.70%)
445.gobmk	518	522 (0.77%)	519 (0.19%)
447.dealII	1294	1301 (0.54%)	1300 (0.46%)
450.soplex	339	345 (1.78%)	345 (1.78%)
453.povray	440	-	451 (2.5%)
456.hmmer	569	-	572 (0.52%)
458.sjeng	620	621 (0.16%)	622 (0.32%)
462.libquantum	474	481 (2.34%)	481 (2.34%)
464.h264ref	872	877 (0.57%)	879 (0.80%)
470.lbm	692	695 (0.43%)	694 (0.28%)
471.omnetpp	781	-	802 (2.6%)
473.astar	544	546 (0.33%)	546(0.33%)
482.sphinx3	945	947 (0.21%)	946 (0.11%)
483.xalanbmk	1325	-	1341(1.2%)

2.8 Related Work

Software Debloating is a well-known attack mitigation scheme which reduces code size and complexity. Rastogi et al. introduced *Cimplifier* [16], an approach for debloating containers by using dynamic analysis for necessary resource identification. *Chisel* [14] debloats programs at a fine-grained level through reinforcement learning. *Trimmer* [53] eliminates unused functionality based on user-provided configuration data. Quanch et al. [15] debloat programs via piece wise compilation and loading. They analyze the program to build a dependency graph of external functions and then only load the required functions as well as remove any library code. Nibbler [17] performs similar library specialization at the binary level. BinTrimmer [54] utilizes abstract interpretation to recover a precise CFG as well as to identify unreachable code and then removing it. Unfortunately, software debloating is not enough to stop CFH. An attacker can still exploit bugs in the remaining code segments and launch code-reuse attacks.

Razor [13] is another post-deployment debloating framework which works at the binary level. It has three components: Tracer, Path finder and Generator. It debloats the binary by utilizing test cases to trace execution paths, then uses four heuristics to find nearby code-paths. Finally, the generator rewrites the binary. Similar to Razor, Binary Control-Flow Trimming [44] uses test traces and later machine learning to explore relevant control-flows. Both of these works are binary based and utilize test traces, where as Ancile works primarily on source code and it depends on the user given seeds to map functionality into code. The main distinction of Ancile over these two works is it introduces seed demonstrated fuzzing to explore relevant code regions. It strengthens the security of an application by not only debloating unused functionalities, but also eliminating invalid targets from the remaining control transfers.

Feature-based software specialization is a different software debloating approach that relies on feature specifications. Jiang et al. [55] propose feature-based software customization for Java bytecode by applying static dataflow analysis and a program slicing technique. But this specialization is done by the developers, whereas JRed [56] performs static analysis and automatically trims unused code from both Java applications and Java Runtime Environment (JRE).

Control-Flow Integrity reduces attack surface by prohibiting illegal control flow transfers from the CFG. After the introduction of the CFI mechanism by Abadi et al. [6] in 2005, the mechanism saw a diverse set of improvements along performance, security, and precision. For a full survey see Burow et. al [22].

LLVM-CFI [12] is a static analysis based CFI approach that is implemented in production compilers with negligible overhead (approximately 1%) [57]. In this approach, each indirect call along with associated targets are clustered into equivalence classes where each indirect call can target any of the addresses within the associated equivalence class. However, due to the reliance on the static analysis, LLVM-CFI struggles with aliasing that results in an over-approximation. An attacker can perform attacks [9, 11, 58, 59] by leading an indirect control flow to a different target within the equivalence class without violating the CFG. LLVM-CFI is seeing wide deployment by Google in Chrome [50] and Android [49].

Recent research efforts improved the precision, and thus security, of CFI. *PittyPat* [28] presents a path-sensitive approach combining hardware-based monitoring and runtime points-to-analysis. It improves preciseness with the cost of additional hardware and performance overhead. In particular, it requires a separate process to monitor and validate the execution traces of the protected process. πCFI [26] starts enforcement of a process with an empty CFG and adds edges dynamically by activating addresses as needed. The security of πCFI depends on an attacker's inability to activate certain edges, otherwise it would provide the same guarantees as a static CFI policy (modulo the complexity of activating the target). VIP [25] adds a measure of control and data-flow sensitivity to the static analysis used by CFI. Ancile achieves greater precision than πCFI or VIP through its functional analysis, and does not require additional system resources like PittyPat.

Existing solutions for control-flow hijacking cannot protect against data-flow attacks and leave the attacker some room. Ancile restricts the application to the bare minimum code required to run the specified functionality and thereby restricts the power of data-only attacks to this exposed functionality. If there is no path to, e.g., `execve` then no modification of the program's memory can bend the control flow to the sensitive function.

2.9 FitJit

2.9.1 Introduction

High-level managed languages like JavaScript are widely used in web pages, server-side applications, browser extensions and even in desktop programs. Such languages rely heavily on run-time systems written in unsafe languages such as C and C++. For improved performance, these run-time systems adopt Just-In Time (JIT) compilation. However, the dynamic nature of managed languages make it hard to extract the accurate equivalent representation in C/C++. Attackers take advantage of this to divert the normal execution of the program in order to perform attacker intended actions. Hence, we need a solution to secure JIT compilation and JITted code against CFH attacks.

2.9.2 Motivating Example

The variables of JavaScript are dynamically typed. Hence, it is important that to ensure that whenever there is a communication between the JavaScript code and underlying C++ runtime system, appropriate type translations are performed. Otherwise, untrusted JavaScript application code can attempt to call and exploit the trusted C++ runtime system. Such bugs often arise since JavaScript engines like V8 (Google Chrome's open-source JavaScript and WebAssembly engine) make it easy for the developers to violate JavaScript's memory and type safety [60].

CVE-2020-6418 is a type confusion vulnerability in V8. Listing 2.2 [61] shows this type confusion attack where an attacker attempts to access data in an unauthorized way, thereby executing the malicious code.

```
ITERATIONS = 10000;
TRIGGER = false;

function f(a, p) {
    return a.pop(Reflect.construct(function()
    {}, arguments, p));}

let a;
let p = new Proxy(Object,{get: function(){
        if (TRIGGER) {a[2] = 1.1;}
         return Object.prototype;}});

for(let i = 0; i < ITERATIONS; i++){
    let isLastIteration = i == ITERATIONS - 1;
    a = [0, 1, 2, 3, 4];
    if (isLastIteration)
        TRIGGER = true;
    print(f(a, p));
}
```

Listing 2.2 Example of a type confusion attack, CVE-2020-6148

2.9.3 Attack Surface

Since the the application code can be easily controlled by an attacker, JIT compilers such as V8 has large attack surface. The most common type of attack is JIT-spraying attack [62]. In such attacks, an attacker takes advantage of variable length of instruction length in the x86 architecture. The attacker crafts an input program with carefully chosen constants. For example, the following code snippet is an xor operation between a variable and a constant 0x3C909090.

var = var $\wedge$ 0x3C909090

Assuming the attacker can control the program counter, she can leverage the variable length instruction features of x86 architecture and change th program counter to point the first 0x90 in the above. In consequence, the next executable instruction will then become a no-op (0x90) instruction. To make the attack more effective, the attacker can spray copies of the same code in memory.

2.9.4 Related Work

RockJIT [63] enforces fine-grained CFI on JIT compiler and coarse-grained CFI on the jitted code with a 14% performance overhead. NaCL-JIT [64] enforces coarse-grained CFI policy which implements software based fault isolation by putting the JIT compiler and jitted code in a sandbox. It enforces aligned chunk CFI and incurs 51% overhead. Both the approaches suffer from high overhead and apply very coarse grained security, making them still bypassable and practically infeasible.

Software diversification based mechanism Librando [65] puts random amount of no-ops in the jitted code as well as replaces instruction sequences that have constant operands with other equivalent instruction sequences. Because of its black-box nature, librando needs to disassemble the jitted code and then re-assemble the new code. Another diversification approach, INSeRT [66] combines randomization of intrinsic elements of machine instructions as well as randomly plants special trapping snippets. Wu et al. [67] proposed Removing IMmediate (RIM) where they eliminates immediate values in the native code. It replaces the original opcode with an immediate operand to a new opcode with register operand. Additionally, it randomizes the arrangement of registers to prohibit register layout prediction. Diversification approaches can make CFH attack harder, but a determined attacker will still be able to bypass the mitigation.

JitSafe [68] narrows the time window of the JIT compiled code in the executable pages and eliminates all the immediate values. JitDefender [69] applies code execution control on the VMs and distinguishes the benign usage of JIT-code from malicious usage of the attacker.

Existing work does not ensure type integrity as well as fine-grained CFI in JIT compilers and jitted code in general, leaving enough space for an attacker to perform CFH attacks. Therefore, we need a strong defense mechanism to safe-guard language boundaries from such attacks.

2.10 Proposed Policy

JavaScript is a dynamically typed language. The object and type representation in JavaScript differs hugely from its underlying run-time system written in C++. We propose a run-time monitor to enforce type integrity and control-flow integrity both in JIT compiler and jitted code.

To enforce type integrity, we propose to explicitly check the type at run-time, whenever object calls are dispatched across language barriers and inside both the JS and C++ world. In order to do so, we build a meta data structure to store object type information (for both the C++ and JS world) and verify accurate type information at run-time.

To enforce fine-grained CFI, we propose a segmented CFI policy. Figure 2.8 shows the three different CFI policies for three different code regions.

- **Policy1:** For C++ code regions, we propose to apply static CFI approaches such as Ancile.

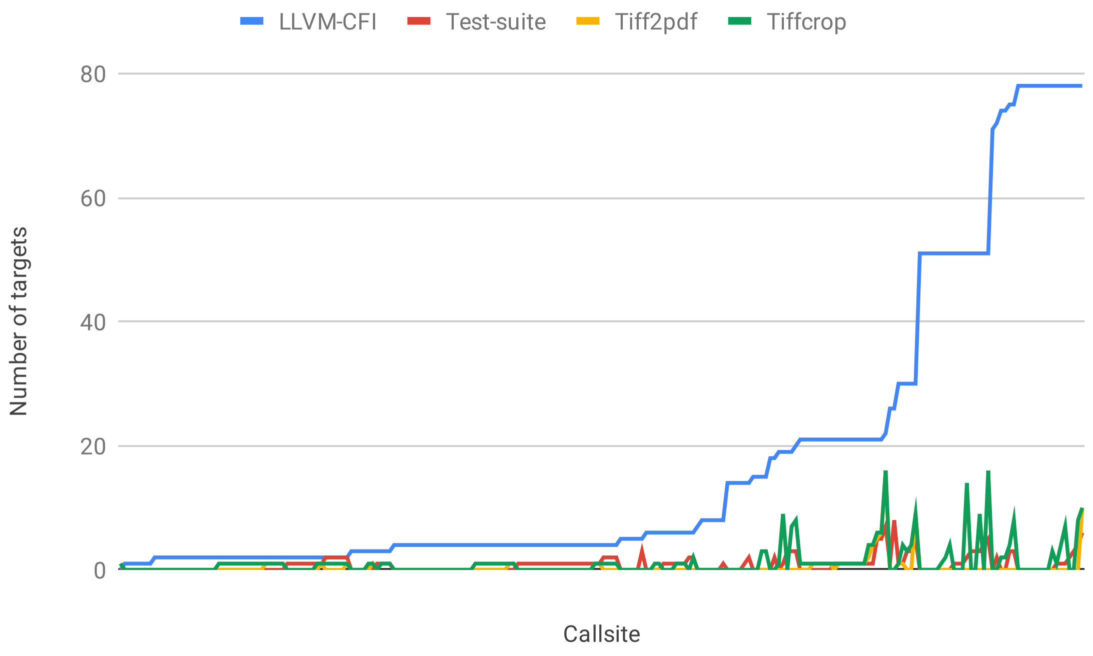
- **Policy2:** For C++ and JavaScript binding layers, we propose mapping one-to-one functions.
- **Policy3:** For JavaScript code base, we propose, callback function based approach. In this technique, we use a v8 wrapper to collect all the function information and enforce integrity via call-back functions.

Security critical application like JIT compiler along with jitted code requires strong defense mechanism against CFH attacks. Therefore, FitJit aims to guarantee strong security in such applications by ensuring type and control-flow integrity.

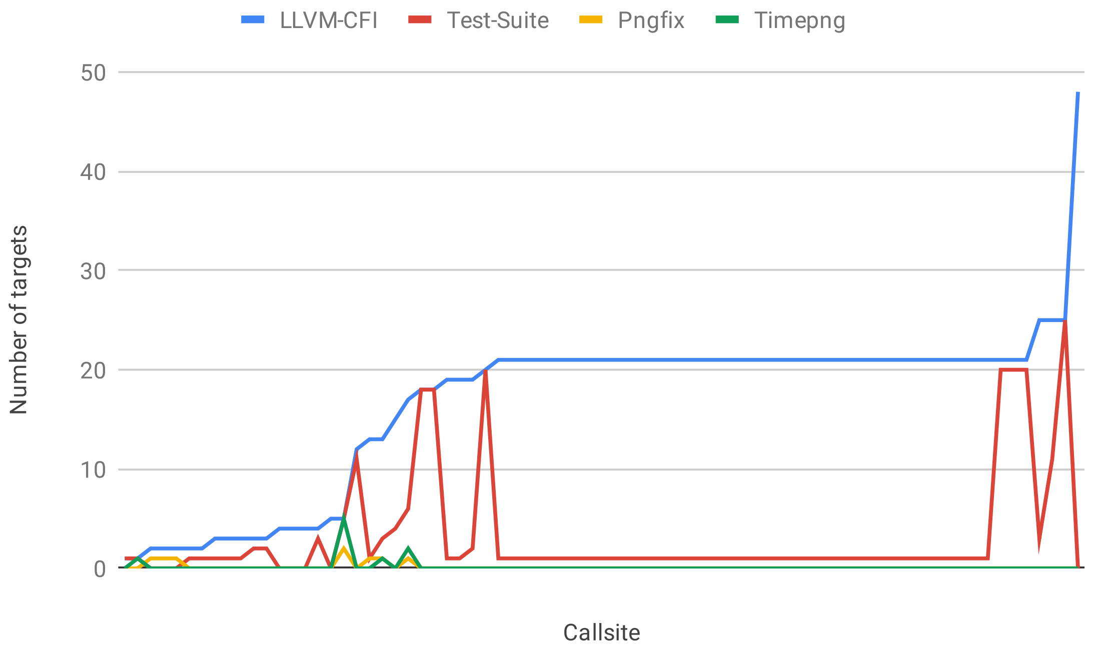
2.11 Conclusion

We present HexVASAN, a code specialization technique through fuzzing. Our case studies show that targeted fuzzing can be used to effectively map user-intended functionalities into relevant code regions. We can then leverage this information to guide debloating and program specialization, reducing the program's attack surface and improving the precision of defenses such as CFI.

We believe that automatically specializing code for particular usage scenarios via fuzzing is a promising new technique for software security. It can achieve greater security than static analysis without requiring extra system resources.

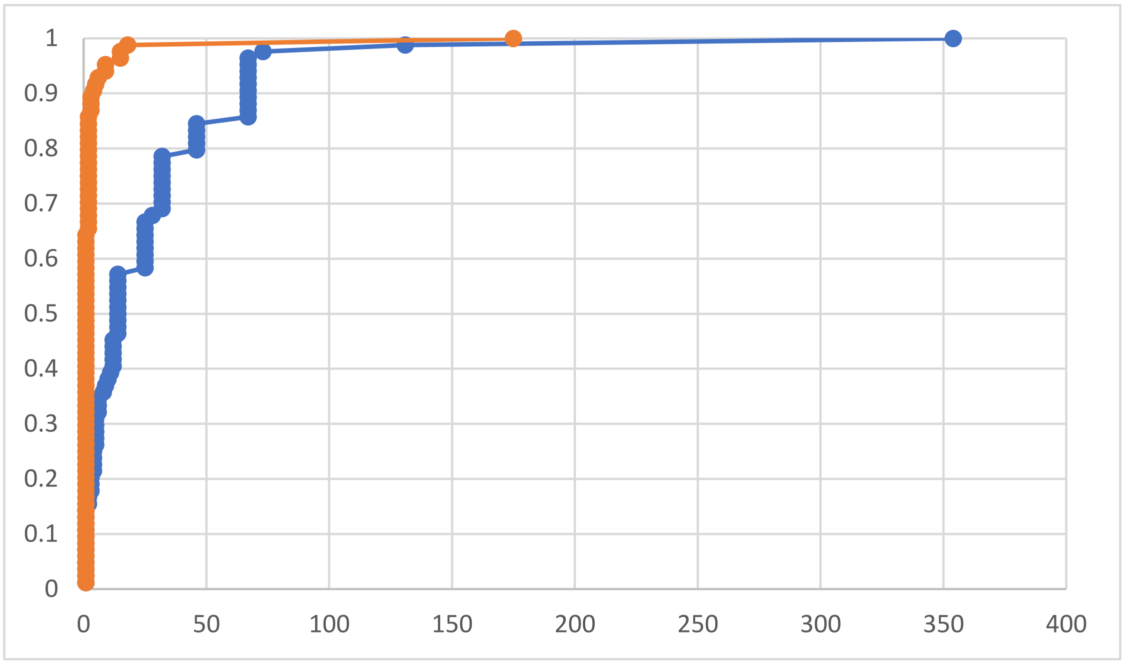


(a) libtiff

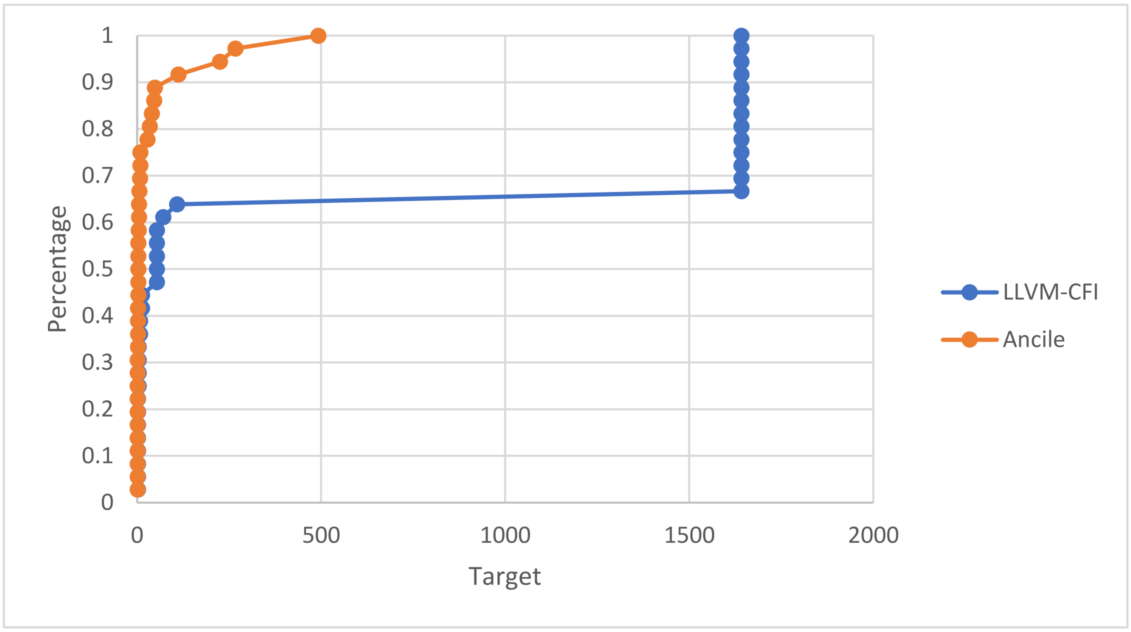


(b) libpng

Figure 2.4. Comparison of number of targets per each callsite at LLVM-CFI and Ancile with specialization in different functionalities for two libraries: libtiff and libpng. For each case study, we analyzed LLVM-CFI and Ancile with three different functionality scenarios: standard test-suite along with two utilities (*tiffcrop* and *tiff2pdf* utilities for libtiff, and *pngfix* and *timepng* utilities for libpng)



(a) 400.perlbench



(b) 445.gobmk

Figure 2.5. Comparison of the cumulative distribution function (CDF)

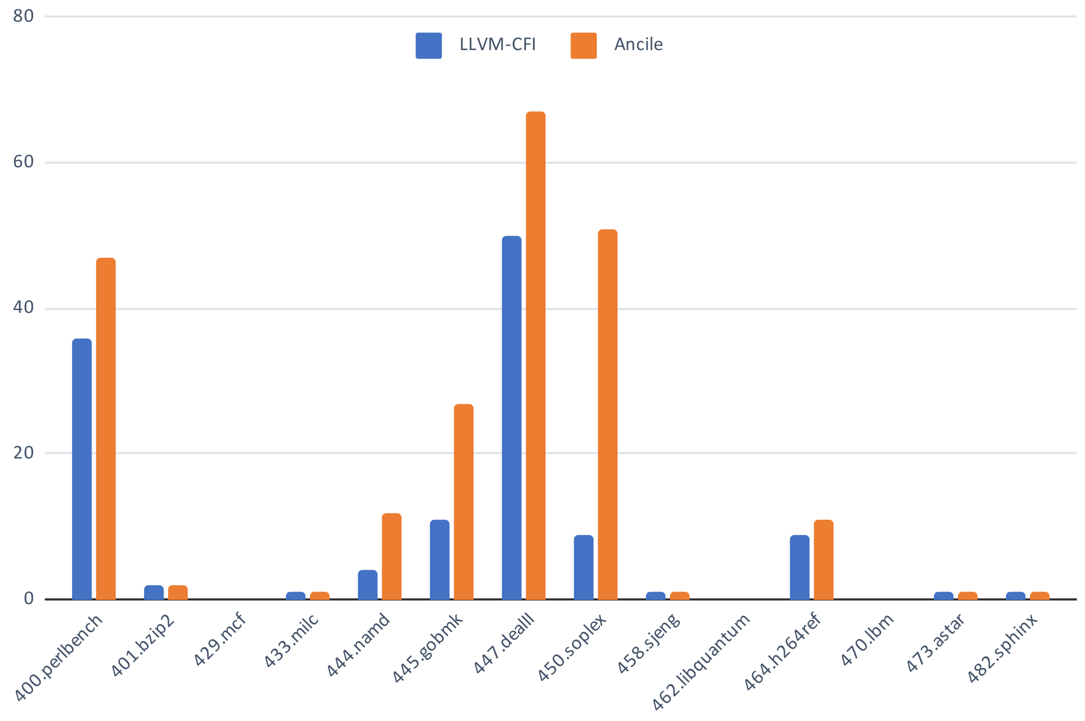


Figure 2.6. Statistics of the number of equivalence classes for SPEC CPU2006 benchmarks.

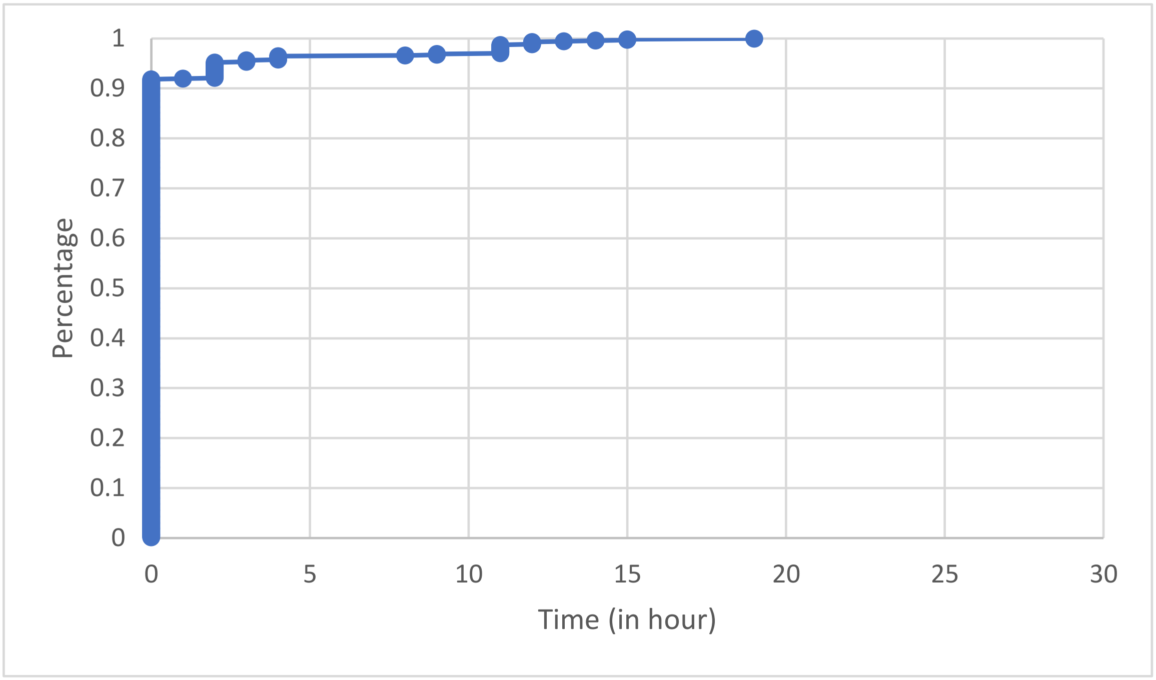


Figure 2.7. Target discovery over the time during application (tcpdump) fuzzing.

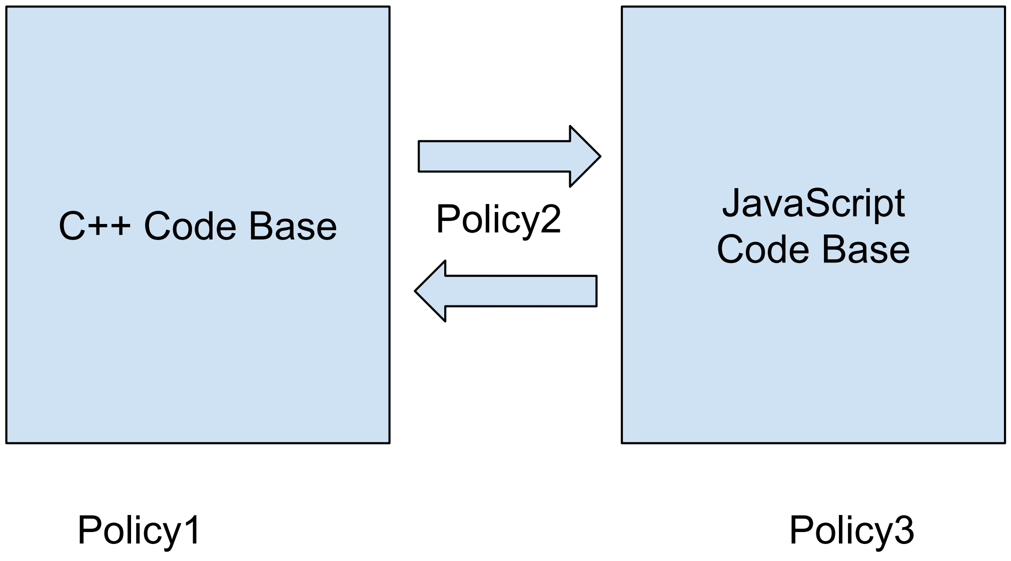


Figure 2.8. Proposed segmented CFI policy for language boundaries

3 HEXVASAN

Programming languages such as C and C++ support variadic functions, i.e., functions that accept a variable number of arguments (e.g., `printf`). While variadic functions are flexible, they are inherently not type-safe. In fact, the semantics and parameters of variadic functions are defined implicitly by their implementation. It is left to the programmer to ensure that the caller and callee follow this implicit specification, without the help of a static type checker. An adversary can take advantage of a mismatch between the argument types used by the caller of a variadic function and the types expected by the callee to violate the language semantics and to tamper with memory. Format string attacks are the most popular example of such a mismatch.

Indirect function calls can be exploited by an adversary to divert execution through illegal paths. CFI restricts call targets according to the function prototype which, for variadic functions, does not include all the actual parameters. However, as shown by our case study, current CFI implementations are mainly limited to non-variadic functions and fail to address this potential attack vector. Defending against such an attack requires a stateful dynamic check.

We present HexVASAN, a compiler based sanitizer to effectively type-check and thus prevent any attack via variadic functions (when called directly or indirectly). The key idea is to record metadata at the call site and verify parameters and their types at the callee whenever they are used at runtime. Our evaluation shows that HexVASAN is (i) practically deployable as the measured overhead is negligible (0.45%) and (ii) effective as we show in several case studies.

3.1 Introduction

In this work, we present a new attack against widely deployed mitigations through a frequently used feature in C/C++ that has so far been overlooked: variadic functions. Variadic functions (such as `printf`) accept a varying number of arguments with varying argument types. To implement variadic functions, the programmer implicitly encodes the argument list in the semantics of the function and has to make sure the caller and callee adhere to this implicit contract. In `printf`, the expected number of arguments and their types are encoded implicitly in the format string, the first argument to the function. Another frequently used scheme iterates through parameters until a condition is reached (e.g., a parameter is `NULL`). Listing 3.1 shows an example of a variadic function. If an adversary can violate the implicit contract between caller and callee, an attack may be possible.

In the general case, it is impossible to enumerate the arguments of a variadic function through static analysis techniques. In fact, their number and types are intrinsic in how the function is defined. This limitation enables (or facilitates) two attack vectors against variadic functions. First, attackers can hijack indirect calls and thereby call variadic functions over control-flow edges that are never taken during any legitimate execution of the program. Variadic functions that are called in this way may interpret the variadic arguments differently than the function for which these arguments were intended, and thus violate the implicit caller-callee contract. CFI countermeasures specifically prevent illegal calls over indirect call edges. However, even the most precise implementations of CFI, which verify the type signature of the targets of indirect calls, are unable to fully stop illegal calls to variadic functions.

A second attack vector involves overwriting a variadic function's arguments directly. Such attacks do not violate the intended control flow of a program and thus bypass all of the widely deployed defense mechanisms. Format string attacks are a prime example of such attacks. If an adversary can control the format string passed to, e.g., `printf`, she can control how all of the following parameters are interpreted, and can potentially leak information from the stack, or read/write to arbitrary memory locations.

The attack surface exposed by variadic functions is significant. We analyzed popular software packages, such as Firefox, Chromium, Apache, CPython, nginx, OpenSSL, Wireshark, the SPEC CPU2006 benchmarks, and the FreeBSD base system, and found that variadic functions are ubiquitous. We also found that many of the variadic function calls in these packages are indirect. We therefore conclude that both attack vectors are realistic threats. The underlying problem that enables attacks on variadic functions is the lack of type checking. Variadic functions generally do not (and cannot) verify that the number and type of arguments they expect matches the number and type of arguments passed by the caller. We present HexVASAN, a compiler-based, dynamic sanitizer that tackles this problem by enforcing type checks for variadic functions at run-time. Each argument that is retrieved in a variadic function is type checked, enforcing a strict contract between caller and callee so that (i) a maximum of the passed arguments can be retrieved and (ii) the type of the arguments used at the callee are compatible with the types passed by the caller. Our mechanism can be used in two operation modes: as a runtime monitor to protect programs against attacks and as sanitizer to detect type mismatches during program testing.

We have implemented HexVASAN on top of the LLVM compiler framework, instrumenting the compiled code to record the types of each argument of a variadic function at the call site and to check the types whenever they are retrieved. Our prototype implementation is light-weight, resulting in negligible (0.45%) overhead for SPEC CPU2006. Our approach is general as we show by recompiling the FreeBSD base system and effective as shown through several exploit case studies (e.g., a format string vulnerability in `sudo`).

```c
#include <stdio.h>
#include <stdarg.h>

int add(int start, ...) {
    int next, total = start;
    va_list list;
    va_start(list, start);
    do {
        next = va_arg(list, int);
        total += next;
    } while (next != 0);
    va_end(list);
    return total;
}

int main(int argc, const char *argv[]) {
    printf("%d\n", add(5, 1, 2, 0));
    return 0;
}
```

Listing 3.1 Example of a variadic function in C. The function `add` takes a non-variadic argument `start` (to initialize an accumulator variable) and a series of variadic `int` arguments that are added until the terminator value `0` is met. The final value is returned.

3.2 Background

Variadic functions are used ubiquitously in C/C++ programs. Here we introduce details about their use and implementation on current systems, the attack surface they provide, and how adversaries can abuse them.

3.2.1 Variadic functions

Variadic functions (such as the `printf` function in the C standard library) are used in C to maximize the flexibility in the interface of a function, allowing it to accept a number of arguments unknown at compile-time. These functions accept a variable number of arguments, which do not necessarily have fixed types. An example of a variadic function is shown in Listing 3.1. The function `add` accepts one mandatory argument (`start`) and a varying number of additional arguments, which are marked by the ellipsis (`...`) in the function definition.

The C standard defines several macros that portable programs may use to access variadic arguments [70]. `stdarg.h`, the header that declares these macros, defines an opaque type, `va_list`, which stores all information required to retrieve and iterate through variadic arguments. In our example, the variable `list` of type `va_list` is initialized using the `va_start` macro. The `va_arg` macro retrieves the next variadic argument from the `va_list`, updating `va_list` to point to the next argument as a side effect. Note that, although the programmer must specify the expected type of the variadic argument in the call to `va_arg`, the C standard does not require the compiler to verify that the retrieved variable is indeed of that type. `va_list` variables must be released using a call to the `va_end` macro so that all of the resources assigned to the list are deallocated.

`printf` is an example of a more complex variadic function which takes a format string as its first argument. This format string implicitly encodes information about the number of arguments and their type. Implementations of `printf` scan through this format string several times to identify all format arguments and to recover the necessary space in the output string for the specified types and formats. Interestingly, arguments do not have to be encoded sequentially but format strings allow out-of-order access to arbitrary arguments. This flexibility is often abused in format string attacks to access arbitrary stack locations.

3.2.2 Variadic functions ABI

The C standard does not define the calling convention for variadic functions, nor the exact representation of the `va_list` structure. This information is instead part of the ABI of the target platform.

x86-64 ABI. The AMD64 System V ABI [71], which is implemented by x86-64 GNU/Linux platforms, dictates that the caller of a variadic function must adhere to the normal calling conventions when passing arguments. Specifically, the first six non-floating point arguments and the first eight floating point arguments are passed through CPU registers. The remaining arguments, if any, are passed on the stack. If a variadic function accepts five mandatory arguments and a variable number of variadic arguments, than all but one of these variadic arguments will be passed on the stack. The variadic function itself moves the arguments into a `va_list` variable using the `va_start` macro. The `va_list` type is defined as follows:

```
typedef struct {
  unsigned int gp_offset;
  unsigned int fp_offset;
  void *overflow_arg_area;
  void *reg_save_area;
} va_list[1];
```

`va_start` allocates on the stack a `reg_save_area` to store copies of all variadic arguments that were passed in registers. `va_start` initializes the `overflow_arg_area` field to point to the first variadic argument that was passed on the stack. The `gp_offset` and `fp_offset` fields are the offsets into the `reg_save_area`. They represent the first unused variadic argument that was passed in a general purpose register or floating point register respectively.

The `va_arg` macro retrieves the first unused variadic argument from either the `reg_save_area` or the `overflow_arg_area`, and either it increases the `gp_offset`/`fp_offset` field or moves the `overflow_arg_area` pointer forward, to point to the next variadic argument.

Other architectures. Other architectures may implement variadic functions differently. On 32-bit x86, for example, all variadic arguments must be passed on the stack (pushed right to left), following the `cdecl` calling convention used on GNU/Linux. The variadic function itself retrieves the first unused variadic argument directly from the stack. This simplifies the implementation of the `va_start`, `va_arg`, and `va_end` macros, but it generally makes it easier for adversaries to overwrite the variadic arguments.

3.2.3 Variadic attack surface

When calling a variadic function, the compiler statically type checks all non-variadic arguments but does not enforce any restriction on the type or number of variadic arguments. The programmer must follow the implicit contract between caller and callee that is only present in the code but never enforced explicitly. Due to this high flexibility, the compiler cannot check arguments statically. This lack of safety can lead to bugs where an adversary achieves control over the callee by modifying the arguments, thereby influencing the interpretation of the passed variadic arguments.

Modifying the argument or arguments that control the interpretation of variadic arguments allows an adversary to change the behavior of the variadic function, causing the callee to access additional or fewer arguments than specified and to change the interpretation of their types.

An adversary can influence variadic functions in several ways. First, if the programmer forgot to validate the input, the adversary may directly control the arguments to the variadic function that control the interpretation of arguments. Second, the adversary may use an arbitrary memory corruption elsewhere in the program to influence the argument of a variadic function.

Variadic functions can be called statically or dynamically. Direct calls would, in theory, allow some static checking. Indirect calls (e.g., through a function pointer), where the target of the variadic function is not known, do not allow any static checking. Therefore, variadic functions can only be protected through some form of runtime checker that considers the constraints of the call site and enforces them at the callee side.

3.2.4 Format string exploits

Format string exploits are a perfect example of corrupted variadic functions. An adversary that gains control over the format string used in `printf` can abuse the `printf` function to leak arbitrary data on the stack or even resort to arbitrary memory corruption (if the pointer to the target location is on the stack). For example, a format string vulnerability in the smbclient utility (CVE-2009-1886) [72] allows an attacker to gain control over the Samba file system by treating a filename as format string. Also, in PHP 7.x before 7.0.1, an error handling function in zend_execute_API.c allows an attacker to execute arbitrary code by using format string specifiers as class name (CVE-2015-8617) [73].

Information leaks are simple: an adversary changes the format string to print the desired information that resides somewhere higher up on the stack by employing the desired format string specifiers. For arbitrary memory modification, an adversary must have the target address encoded somewhere on the stack and then reference the target through the `%n` modifier, writing the number of already written bytes to that memory location.

The GNU C standard library (*glibc*) enforces some protection against format string attacks by checking if a format string is in a writable memory area [74]. For format strings, the *glibc* `printf` implementation opens `/proc/self/maps` and scans for the memory area of the format string to verify correct permissions. Moreover, a check is performed to ensure that all arguments are consumed, so that no out-of-context stack slots can be used in the format string exploit. These defenses stop some attacks but do not mitigate the underlying problem that an adversary can gain control over the format string. Note that this heavyweight check is only used if the format string argument *may* point to a writable memory area at compile time. An attacker may use memory corruption to redirect the format string pointer to an attacker-controlled area and fall back to a regular format string exploit.

3.3 Threat model

Programs frequently use variadic functions, either in the program itself or as part of a shared library (e.g., `printf` in the C standard library). We assume that the program contains an arbitrary memory corruption, allowing the adversary to modify the arguments to a variadic function and/or the target of an indirect function call, targeting a variadic function.

Our target system deploys existing defense mechanisms like DEP, ASLR, and a strong implementation of CFI, protecting the program against code injection and control-flow hijacking. We assume that the adversary cannot modify the metadata of our runtime monitor. Protecting metadata is an orthogonal engineering problem and can be solved through, e.g., masking (`and`-ing every memory access), segmentation (for x86-32), protecting the memory region [75], or probabilistically by randomizing the location of sensitive data. Our threat model is a realistic scenario for current attacks and defenses.

3.4 HexVASAN design

HexVASAN monitors calls to variadic functions and checks for type violations. Since the semantics of how arguments should be interpreted by the function are intrinsic in the logic of the function itself, it is, in general, impossible to determine the number and type of arguments a certain variadic function accepts. For this reason, HexVASAN instruments the code generated by the compiler so that a check is performed at runtime. This check ensures that the arguments consumed by the variadic function match those passed by the caller.

The high level idea is the following: HexVASAN records metadata about the supplied argument types at the call site and verifies that the extracted arguments match in the callee. The number of arguments and their types is always known at the call site and can be encoded efficiently. In the callee this information can then be used to verify individual arguments when they are accessed. To implement such a sanitizer, we must design a metadata store, a pass that instruments call sites, a pass that instruments callers, and a runtime library that manages the metadata store and performs the run-time type verification. Our runtime library aborts the program whenever a mismatch is detected and generates detailed information about the call site and the mismatched arguments.

3.4.1 Analysis and Instrumentation

We designed HexVASAN as a compiler pass to be run in the compilation pipeline right after the C/C++ frontend. The instrumentation collects a set of statically available information about the call sites, encodes it in the LLVM module, and injects calls to our runtime to perform checks during program execution.

Figure 3.1 provides an overview of the compilation pipeline when HexVASAN is enabled. Source files are first parsed by the C/C++ frontend which generates the intermediate representation on which our instrumentation runs. The normal compilation then proceeds, generating instrumented object files. These object files,

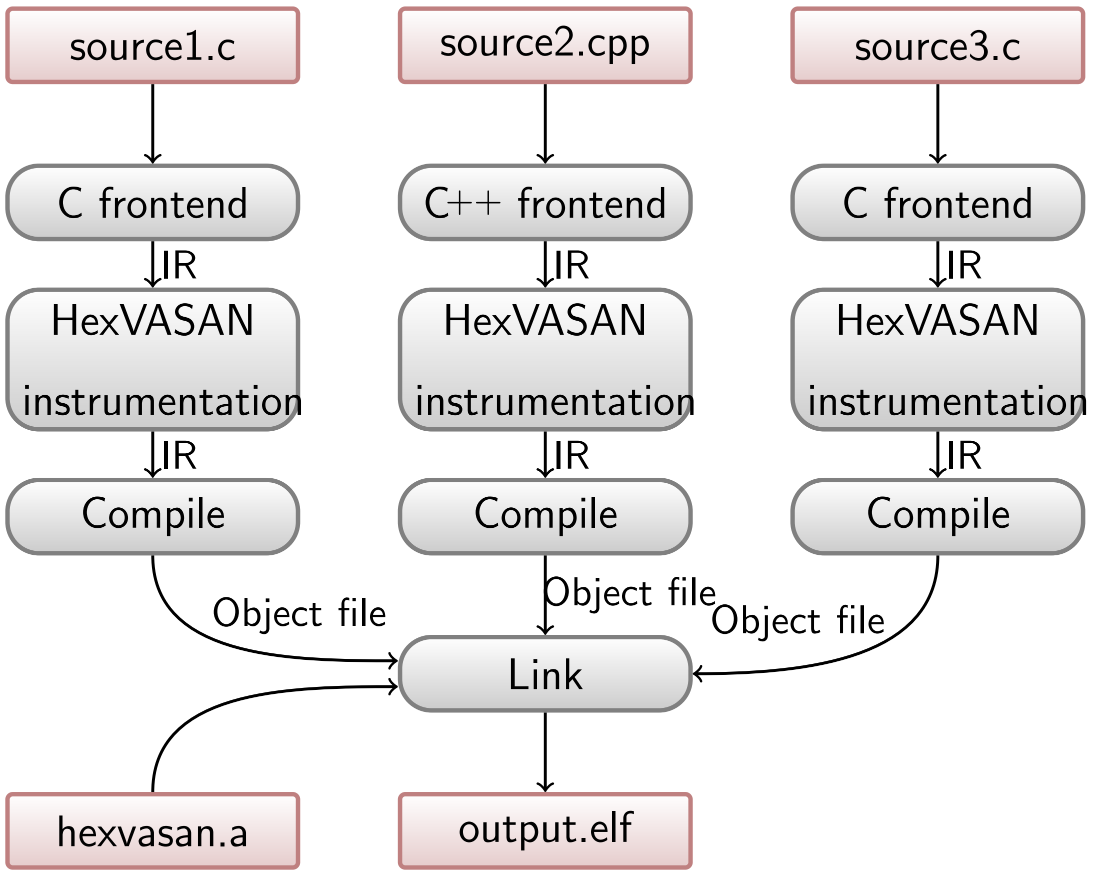


Figure 3.1. Overview of the HexVASAN compilation pipeline. The HexVASAN instrumentation runs right after the C/C++ frontend, while its runtime library, `hexvasan.a`, is merged into the final executable at link time.

along with the HexVASAN runtime library, are then passed to the linker, which creates the instrumented program binary.

3.4.2 Runtime support

The HexVASAN runtime augments every `va_list` in the original program with the type information generated by our instrumentation pass, and uses this type information to perform run-time type checking on any variadic argument accessed through `va_arg`. By managing the type information in a metadata store, and by maintaining a mapping between `va_lists` and their associated type information, HexVASAN remains fully compatible with the platform ABI. This design also supports interfacing between instrumented programs and non-instrumented libraries.

The HexVASAN runtime manages the type information in two data structures. The core data structure, called the *variadic list map* (VLM), associates `va_list` structures with the type information produced by our instrumentation, and with a counter to track the index of the last argument that was read from the list. A second data structure, the *variadic call stack* (VCS), allows callers of variadic functions to store type information of variadic arguments until the callee initializes the `va_list`.

Each variadic call site is instrumented with a call to `pre_call`, that prepares the information about the call site (a *variadic call site descriptor* or VCSD), and a call to `post_call`, that cleans it up. For each variadic function, the `va_start` calls are instrumented with `list_init`, while `va_copy`, whose purpose is to clone a `va_list`, is instrumented through `list_copy`. The two run-time functions will allocate the necessary data structures to validate individual arguments. Calls to `va_end` are instrumented through `list_end` to free up the corresponding data structures.

Algorithm 1 summarizes the two phases of our analysis and instrumentation pass. The first phase identifies all the calls to variadic functions (both direct and indirect). Note that identifying indirect calls to variadic functions is straight-forward in a compiler framework since, even if the target function is not statically known, its type is.

```
input: a module $m$
/* Phase 1                                                    */
foreach function $f$ in module $m$ do
    foreach variadic call $c$ with $n$ arguments in $f$ do
        vcsd.count $\leftarrow n$;
        foreach argument $a$ of type $t$ do
            vcsd.args.push($t$);
        end
        emit call to pre_call($vcsd$) before $c$;
        emit call to post_call() after $c$;
    end
end
/* Phase 2                                                    */
foreach function $f$ in module $m$ do
    foreach call $c$ to va_start($list$) do
        emit call to list_init(&$list$) after $c$;
    end
    foreach call $c$ to va_copy($dst, src$) do
        emit call to list_copy(&$dst$, &$src$) after $c$;
    end
    foreach call $c$ to va_end($list$) do
        emit call to list_free(&$list$) after $c$;
    end
    foreach call $c$ to va_arg($list, type$) do
        emit call to check_arg(&$list, type$) before $c$;
    end
end
```

Algorithm 1: The instrumentation process.

Then, all the parameters passed by that specific call site are inspected and recorded, along with their type in a dedicated VCSD which is stored in read-only global data. At this point, a call to `pre_call` is injected before the variadic function call (with the newly created VCSD as a parameter) and, symmetrically, a call to `post_call` is inserted after the call site.

The second phase identifies all calls to `va_start` and `va_copy`, and consequently, the `va_list` variables in the program. Uses of each `va_list` variable are inspected in an architecture-specific way. Once all uses are identified, we inject a call to `check_arg` before dereferencing the argument (which always resides in memory).

3.4.3 Challenges and Discussion

When designing a variadic function call sanitizer, several issues have to be considered. We highlight details about the key challenges we encountered.

Multiple `va_lists`. Functions are allowed to create multiple `va_lists` to access the same variadic arguments, either through `va_start` or `va_copy` operations. HexVASAN handles this by storing a VLM entry for each individual `va_list`.

Passing `va_lists` as function arguments. While uncommon, variadic functions are allowed to pass the `va_lists` they create as arguments to non-variadic functions. This allows non-variadic functions to access variadic arguments of functions higher in the call stack. Our design takes this into account by maintaining a list map (VLM) and by instrumenting all `va_arg` operations, regardless of whether or not they are in a variadic function.

Multi-threading support. Multiple threads are supported by storing our per-thread runtime state in thread-local variables that are supported on all major operating systems.

Metadata format. The main goals in designing the metadata encoding is to make it lean and straight-forward to use. We opted for having a constant data structure per variadic call site, the VCSD, that holds the number of arguments and a pointer to an array of integers identifying their type. The `check_arg` function therefore only performs two memory accesses, the first to load the number of arguments and the second for the type of the argument currently being checked.

To uniquely identify the data types with an integer, we decided to build a hashing function (described in Algorithm 2) using a set of fixed identifiers for primitive data types and hashing them in different ways depending on how they are aggregated (pointers, `union`, or `struct`). This approach has the advantage of being deterministic across compilation units, removing the need for keeping a global map of type-unique id pairs. Note that (unlikely) hash collisions only result in two different types being accepted as equal. Due to the information loss during the translation from C/C++ to LLVM IR, our type system does not distinguish between signed and unsigned types. The required metadata is static and immutable and we mark it as read-only, protecting it from modification. However, the VCS still needs to be protected through other mechanisms.

Handling floating point arguments. In x86-64 ABI, floating point and non-floating point arguments are handled differently. In case of floating point arguments, the first eight arguments are passed in the floating point registers whereas in case of non-floating point the first six are passed in general-purpose registers. HexVASAN handles both argument types.

Support for aggregate data types. According to AMD64 System V ABI, the caller unpacks the fields of the aggregate data types (structs and unions) if the arguments fit into registers. This makes it hard to distinguish between composite types and regular types – if unpacked they are indistinguishable on the callee side from arguments of these types. HexVASAN supports aggregate data types even if the caller unpacks them.

Attacks preserving number and type of arguments. Our mechanism prevents attacks that change the number of arguments or the types of individual arguments. Format string attacks that only change one modifier can therefore be detected through the type mismatch even if the total number of arguments remains unchanged.

```
input  : a type $t$ and an initial hash value $h$
output: the final hash value $h$
$h$ = hash($h$, typeID($t$));
switch typeID($t$) do
    case AggregateType do
        /* union, struct and pointer                                   */
        foreach $c$ in componentTypes($t$) do
            $h$ = hashType($c$, $h$);
        end
    case FunctionType do
        $h$ = hashType(returnType($t$), $h$);
        foreach $a$ in argTypes($t$) do
            $h$ = hashType($a$, $h$);
        end
    end
end
$h$ = hash($h$, typeID($t$));
return h
```

Algorithm 2: Algorithm describing the type hashing function *hashType*. *typeID* returns an unique identifier for each basic type (e.g., 32-bit integer, `double`), type of aggregate type (e.g., `struct`, `union`...) and functions. *hash* is a simple hashing function combining two integers. *componentTypes* returns the components of an aggregate type, *returnType* the return type of a function prototype and *argTypes* the type of its arguments.

Non-variadic calls to variadic functions. Consider the following code snippet:

```
typedef void (*non_variadic)(int, int);

void variadic(int, ...) { /* ... */ }

int main() {
  non_variadic function_ptr = variadic;
  function_ptr(1, 2);
}
```

In this case, the function call in `main` to `function_ptr` appears to the compiler as a non-variadic function call, since the type of the function pointer is not variadic. Therefore, our pass will not instrument the call site, leading to potential errors.

To handle such (rare) situations appropriately, we would have to instrument all non-variadic call sites too, leading to an unjustified overhead. Moreover, the code above represents *undefined behavior* in C [76, 6.3.2.3p8] and C++ [77, 5.2.10p6], and might not work on certain architectures where the calling convention for variadic and non-variadic function calls are not compatible. The GNU C compiler emits a warning when a function pointer is cast to a different type, therefore we require the developer to correct the code before applying HexVASAN.

Central management of the global state. To allow the HexVASAN runtime to be linked into the base system libraries, such as the C standard library, we made it a static library. Turning the runtime into a shared library is possible, but would prohibit its use during the early process initialization – until the dynamic linker has processed all of the necessary relocations. Our runtime therefore either needs to be added solely to the C standard library (so that it is initialized early in the startup process) or the runtime library must carefully use weak symbols to ensure that each symbol is only defined once if multiple libraries are compiled with our countermeasure.

C++ exceptions and `longjmp`. If an exception is raised while executing a variadic function (or one of its callees), the variadic function may not get a chance to clean up the metadata for any `va_lists` it has initialized, nor may the caller of this variadic function get the chance to clean up the type information it has pushed onto the VCS. Other functions manipulating the thread's stack directly, such as `longjmp`, present similar issues.

C++ exceptions can be handled by modifying the LLVM C++ frontend (i.e., `clang`) to inject an object with a lifetime spanning from immediately before a variadic function call to immediately after. Such an object would call `pre_call` in its constructor and `post_call` in the destructor, leveraging the exception handling mechanism to make HexVASAN exception-safe. Functions like `longjmp` can be instrumented to purge the portions of HexVASAN's data structures that correspond to the discarded stack area. We did not observe any such calls in practice and leave the implementation of handling exceptions and `longjump` across variadic functions as future engineering work.

3.5 Implementation

We implemented HexVASAN as a sanitizer for the LLVM compiler framework [78], version 3.9.1. We have chosen LLVM for its robust features on analyzing and transforming arbitrary programs as well as extracting reliable type information. The sanitizer can be enabled from the C/C++ frontend (`clang`) by providing the `-fsanitize=vasan` parameter at compile-time. No annotations or other source code changes are required for HexVASAN. Our sanitizer does not require visibility of whole source code (see Section 3.4.3), but works on individual compilation units. Therefore link-time optimization (LTO) is not required and thus fits readily into existing build systems.

HexVASAN consists of two components: a static instrumentation pass and a runtime library. The static instrumentation pass works on LLVM IR, adding the necessary instrumentation code to all variadic functions and their callees. The support library is statically linked to the program and, at run-time, checks the number and type of variadic arguments as they are used by the program. In the following we describe the two components in detail.

Static instrumentation. The implementation of the static instrumentation pass follows the description in Section 3.4. We first iterate through all functions, looking for `CallInst` instructions targeting a variadic function (either directly or indirectly), then we inspect them and create for each one of them a read-only `GlobalVariable` of type `vcsd_t`. As shown in Listing 3.2, `vcsd_t` is composed by an unsigned integer representing the number of arguments of the considered call site and a pointer to an array (another `GlobalVariable`) with an integer element for each argument of `type_t`. `type_t` is an integer uniquely identifying a data type obtained using the *hashType* function presented in Algorithm 2. At this point a call to `pre_call` is injected before the call site, with the newly create VCSD as a parameter, and a call to `post_call` is injected after the call site.

During the second phase, we first identify all `va_start`, `va_copy`, and `va_end` operations in the program. In the IR code, these operations appear as calls to the LLVM intrinsics `llvm.va_start`, `llvm.va_copy`, and `va_end`. We instrument the operations with calls to our runtime's `list_init`, `list_copy`, and `list_free` functions respectively. We then proceed to identify `va_arg` operations. Although the LLVM IR has a dedicated `va_arg` instruction, it is not used on any of the platforms we tested. The `va_list` is instead accessed directly. Our identification of `va_arg` is therefore platform-specific. On x86-64, our primary target, we identify `va_arg` by recognizing accesses to the `gp_offset` and `fp_offset` fields in the x86-64 version of the `va_list` structure (see Section 3.2.2). The `fp_offset` field is accessed whenever the program attempts to retrieve a floating point argument from the list. The `gp_offset` field is accessed to retrieve any other types of variadic arguments. We insert a call to our runtime's `check_arg` function before the instruction that accesses this field.

Listing 3.2 shows (in simplified C) how the code in Listing 3.1 would be instrumented by our sanitizer.

```
struct vcsd_t { unsigned count; type_t *args; };
thread_local stack<vcsd_t *> vcs;
thread_local map<va_list *, pair<vcsd_t *, unsigned>> vlm;
void pre_call(vcsd_t *arguments) { vcs.push_back(arguments); }
void post_call() { vcs.pop_back(); }
void list_init(va_list *list_ptr) { vlm[list_ptr] = { vcs.top(), 0 }; }
void list_free(va_list *list_ptr) { vlm.erase(list_ptr); }
void check_arg(va_list *list_ptr, type_t type) {
  pair<vcsd_t *, unsigned> &args = vlm[list_ptr];
  unsigned index = args.second++;
  assert(index < args.first->count);
  assert(args.first->args[index] == type);
}
int add(int start, ...) {
  va_start(list, start);
  list_init(&list);
  do {
    check_arg(&list, typeid(int));
    total += va_arg(list, int);
  } while (next != 0);
  va_end(list);
  list_free(&list);
}
const vcsd_t main_add_vcsd = {
  .count = 3,
  .args = {typeid(int), typeid(int), typeid(int)}
};
int main(int argc, const char *argv[]) {
  pre_call(&main_add_vcsd);
  int result = add(5, 1, 2, 0);
  post_call();
  printf("%d\n", result);
}
```

Listing 3.2 Simplified C++ representation of the instrumented code for Listing 3.1.

```
Error: Type Mismatch
Index is 1
Callee Type : 43 (32-bit Integer)
Caller Type : 15 (Pointer)
Backtrace:
[0] 0x4019ff <__vasan_backtrace+0x1f> at test
[1] 0x401837 <__vasan_check_arg+0x187> at test
[2] 0x8011b3afa <__vfprintf+0x20fa> at libc.so.7
[3] 0x8011b1816 <vfprintf_l+0x86> at libc.so.7
[4] 0x801200e50 <printf+0xc0> at libc.so.7
[5] 0x4024ae <main+0x3e> at test
[6] 0x4012ff <_start+0x17f> at test
```

Listing 3.3 Error message reported by HexVASAN.

Dynamic variadic type checking. plain C code, as this allows it to be linked into the standard C library without introducing a dependency to the standard C++ library. The VCS is implemented as a thread-local stack, and the VLM as a thread-local hash map. The `pre_call` and `post_call` functions push and pop type information onto and from the VCS. The `list_init` function inserts a new entry into the VLM, using the top element on the stack as the entry's type information and initializing the counter for consumed arguments to 0.

`check_arg` looks up the type information for the `va_list` being accessed in the VLM and checks if the requested argument exists (based on the counter of consumed arguments), and if its type matches the one provided by the caller.

If either of these checks fails, execution is aborted, and the runtime will generate an error message such as the one shown in Listing 3.3. As a consequence, the pointer to the argument is never read or written, since the pointer to it is never dereferenced.

3.6 Evaluation

In this section we present a case study on variadic function based attacks against state-of-the-art CFI implementations. Next, we evaluate the effectiveness of HexVASAN as an exploit mitigation technique. Then, we evaluate the overhead introduced by our HexVASAN prototype implementation on the SPEC CPU2006 integer (CINT2006) benchmarks as well as HexVASAN hardened Firefox using standard JavaScript benchmarks. We also evaluate how widespread the usage of variadic functions is in SPEC CPU2006 and in Firefox 51.0.1, Chromium 58.0.3007.0, Apache 2.4.23, CPython 3.7.0, nginx 1.11.5, OpenSSL 1.1.1, Wireshark 2.2.1, and the FreeBSD 11.0 base system.

Note that, along with testing the above mentioned software, we also developed our internal set of regression tests, consisting of a suite of simple programs whose goal is to assess that our sanitizer can catch problematic situations and benign correctly calls to variadic functions. The test suite explores corner cases, including trying to access arguments that have not been passed and trying to access them using a type different from the actual one.

3.6.1 Case study: CFI effectiveness

One of the attack scenarios we envision is that an attacker controls the target of an indirect call site. If the intended target of the call site was a variadic function, the attacker could illegally call a different variadic function that expects different variadic arguments than the intended target (yet shares the types for all non-variadic arguments). If the intended target of the call site was a non-variadic function, the attacker could call a variadic function that interprets some of the intended target's arguments as variadic arguments.

All existing CFI mechanisms allow such attacks to some extent. The most precise CFI mechanisms, which rely on function prototypes to classify target sets (e.g., LLVM-CFI, piCFI, or VTV) will allow all targets with the same prototype, possibly

	Actual target							
Intended target	Prototype	A.T.?	LLVM-CFI	pi-CFI	CCFI	VTV	CFG	HexVASAN
Variadic	Same	Yes	✗	✗	✗	✗	✗	✓
		No	✗	✓	✗	✗	✗	✓
	Different	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✗	✗	✗	✓
Non-variadic	Same	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✗	✗	✗	✓
	Different	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✓	✗	✗	✓
Original	Overwritten Arguments		✗	✗	✗	✗	✗	✓

Table 3.1.
Detection coverage for several types of illegal calls to variadic functions. ✓ indicates detection, ✗ indicates non-detection. “A.T.” stands for *address taken*.

restricting to the subset of functions whose addresses are taken in the program. This is problematic for variadic functions, as only non-variadic types are known statically. For example, if a function of type `int (*)(int, ...)` is expected to be called from an indirect call site, then precise CFI schemes allow calls to all other variadic functions of that type, even if those other functions expect different types for the variadic arguments.

A second way to attack variadic functions is to overwrite their arguments directly. This happens, for example, in format string attacks, where an attacker can overwrite the format string to cause misinterpretation of the variadic arguments. HexVASAN detects both of these attacks when the callee attempts to retrieve the variadic arguments using the `va_arg` macro described in Section 3.2.1. Checking and enforcing the correct types for variadic functions is only possible at runtime and any sanitizer must resort to run-time checks to do so. CFI mechanisms must therefore be extended with a HexVASAN-like mechanism to detect violations. To show that our tool can complement CFI, we create test programs containing several variadic functions and one non-variadic function. The definitions of these functions are shown below.

```
int sum_ints(int n, ...);
int avg_longs(int n, ...);
int avg_doubles(int n, ...);
void print_longs(int n, ...);
void print_doubles(int n, ...);
int square(int n);
```

This program contains one indirect call site from which only the `sum_ints` function can be called legally, and one indirect call site from which only the `square` function can be legally called. We also introduce a memory corruption vulnerability which allows us to override the target of both indirect calls.

We constructed the program such that `sum_ints`, `avg_longs`, `print_longs`, and `square` are all address-taken functions. The `avg_doubles` and `print_doubles` functions are not address-taken.

Functions `avg_longs`, `avg_doubles`, `print_longs`, and `print_doubles` all expect different variadic argument types than function `sum_ints`. Functions `sum_ints`, `avg_longs`, `avg_doubles`, and `square` do, however, all have the same non-variadic prototype (`int` ↪ `(*)(int)`).

We compiled six versions of the test program, instrumenting them with, respectively, HexVASAN, LLVM 3.9 Forward-Edge CFI [23], Per-Input CFI [26], CCFI [79], GCC 6.2's VTV [23] and Visual C++ Control Flow Guard [80]. In each version, we first built an attack involving a variadic function, by overriding the indirect call sites with a call to each of the variadic functions described above. We then also tested overwriting the arguments of the `sum_ints` function, without overwriting the indirect call target. Table 3.1 shows the detection results.

LLVM Forward-Edge CFI allows calls to `avg_longs` and `avg_doubles` from the `sum_ints` indirect call site because these functions have the same static type signature as the intended call target. This implementation of CFI does not allow calls to variadic functions from non-variadic call sites, however.

CCFI only detects calls to `print_doubles`, a function that is not address-taken and has a different non-variadic prototype than `square`, from the `square` call site. It allows all of the other illegal calls.

GCC VTV, and Visual C++ CFG allow all of the illegal calls, even if the non-variadic type signature does not match that of the intended call target.

pi-CFI allows calls to the `avg_longs` function from the `sum_ints` indirect call site. `avg_longs` is address-taken and it has the same static type signature as the intended call target. pi-CFI does not allow illegal calls to non-address-taken functions or functions with different static type signatures. pi-CFI also does not allow calls to variadic functions from non-variadic call sites.

All implementations of CFI allow direct overwrites of variadic arguments, as long as the original control flow of the program is not violated.

3.6.2 Exploit Detection

To evaluate the effectiveness of our tool as a real-world exploit detector, we built a HexVASAN-hardened version of `sudo` 1.8.3. `sudo` allows authorized users to execute shell commands as another user, often one with a high privilege level on the system. If compromised, `sudo` can escalate the privileges of non-authorized users, making it a popular target for attackers. Versions 1.8.0 through 1.8.3p1 of `sudo` contained a format string vulnerability (CVE-2012-0809) that allowed exactly such a compromise. This vulnerability could be exploited by passing a format string as the first argument (`argv[0]`) of the `sudo` program. One such exploit was shown to bypass ASLR, DEP, and glibc's FORTIFY_SOURCE protection [81].

Although it is `sudo` itself that calls the format string function (`fprintf`), HexVASAN can only detect the violation on the callee side. We therefore had to build hardened versions of not just the `sudo` binary itself, but also the C library. We chose to do this on the FreeBSD platform, as its standard C library can be easily built using LLVM, and HexVASAN therefore readily fits into the FreeBSD build process. As expected, HexVASAN does detect any exploit that triggers the vulnerability, producing the error message shown in Listing 3.4.

3.6.3 Prevalence of variadic functions

To collect data about variadic functions in real software, we extended our instrumentation mechanism to also collect statistics about variadic functions and calls to variadic functions. As shown in Table 3.2, for each program, we collect:

```
$ ln -s /usr/bin/sudo %x%x%x%x
$ ./%x%x%x%x -D9 -A
--------------------------
Error: Index greater than Argument Count
Index is 1
Backtrace:
[0] 0x4053bf <__vasan_backtrace+0x1f> at sudo
[1] 0x405094 <__vasan_check_index+0xf4> at sudo
[2] 0x8015dce24 <__vfprintf+0x2174> at libc.so
[3] 0x8015dac52 <vfprintf_l+0x212> at libc.so
[4] 0x8015daab3 <vfprintf_l+0x73> at libc.so
[5] 0x40bdaf <sudo_debug+0xdf> at sudo
[6] 0x40ada3 <main+0x6c3> at sudo
[7] 0x40494f <_start+0x17f> at sudo
```

Listing 3.4 Exploit detection in sudo.

Calls sites. The number of function calls targeting variadic functions. We report the total number and how many of them are indirect, since they are of particular interest for an attack scenario where the adversary can override a function pointer.

Variadic functions. The number of variadic functions. We report their total number and how many of them have their address taken, since CFI mechanism cannot prevent functions with their address taken from being reachable from indirect call sites.

Variadic prototypes. The number of distinct variadic function prototypes in the program.

Functions-per-prototype. The average number of variadic functions sharing the same prototype. This measures how many targets are available, on average, for each indirect call sites targeting a specific prototype. In practice, this the average number of permitted destinations for an indirect call site in the case of a perfect CFI implementation. We report this value both considering all the variadic functions and only those whose address is taken.

Interestingly, each benchmark we analyzed contains calls to variadic functions and several programs (Firefox, OpenSSL, perlbench, gcc, povray, and hmmer) even contain indirect calls to variadic functions. In addition to *calling* variadic functions, each benchmark also *defines* at least several variadic functions with up to 421 for Firefox, 794 for Chromium, 1368 for FreeBSD, 469 for Wireshark, and 382 for CPython. Variadic functions are therefore prevalent and used ubiquitously in software. Adversaries have plenty of opportunities to modify these calls and to attack the implicit contract between caller and callee. The compiler is unable to enforce any static safety guarantees when calling these functions, either for the number of arguments, nor their types. In addition, many of the benchmarks have variadic functions that are called indirectly, often with their address being taken. Looking at Firefox, a large piece of software, the numbers are even more staggering with several thousand indirect call sites that target variadic functions and 241 different variadic prototypes.

```
static sEnumBuilder _EtherMessageKind( "EtherMessageKind",
    JAM_SIGNAL, "JAM_SIGNAL",
    ETH_FRAME, "ETH_FRAME",
    ETH_PAUSE, "ETH_PAUSE",
    ETHCTRL_DATA, "ETHCTRL_DATA",
    ETHCTRL_REGISTER_DSAP, "ETHCTRL_REGISTER_DSAP",
    ETHCTRL_DEREGISTER_DSAP, "ETHCTRL_DEREGISTER_DSAP",
    ETHCTRL_SENDPAUSE, "ETHCTRL_SENDPAUSE",
    0, NULL
);
```

Listing 3.5 Variadic violation in omnetpp.

The prevalence of variadic functions leaves both a large attack surface for attackers to either redirect variadic calls to alternate locations (even if defense mechanisms like CFI are present) or to modify the arguments so that callees misinterpret the supplied arguments (similar to extended format string attacks).

In addition, the compiler has no insight into these functions and cannot statically check if the programmer supplied the correct parameters. Our sanitizer identified three bugs in `omnetpp`, one of the SPEC CPU2006 benchmarks that implements a discrete event simulator. The benchmark calls a variadic functions with a mismatched type, where it expects a `char *` but receives a NULL, which has type `void *`. Listing 3.5 shows the offending code.

We also identified a bug in SPEC CPU2006's `perlbench`. This benchmark passes the result of a substraction of two character pointers as an argument to a variadic function. At the call site, this argument is a machine word-sized integer (i.e., 64-bits integer on our test platform). The callee truncates this argument to a 32-bit integer by calling `va_arg(list, int)`. HexVASAN reports this (likely unintended) truncation as a violation.

3.6.4 Firefox

We evaluate the performance of HexVASAN by instrumenting Firefox (51.0.1) and using three different browser benchmark suites: Octane, JetStream, and Kraken. Table 3.3 shows the comparison between the HexVASAN instrumented Firefox and native Firefox. To reduce variance between individual runs, we averaged fifteen runs for each benchmark (after one warmup run). For each run we started Firefox, ran the benchmark, and closed the browser. HexVASAN incurs only 1.08% and 1.01% overhead for Octane and JetStream respectively and speeds up around 0.01% for Kraken. These numbers are indistinguishable from measurement noise.

3.6.5 SPEC CPU2006

We measured HexVASAN's run-time overhead by running the SPEC CPU2006 integer (CINT2006) benchmarks on an Ubuntu 14.04.5 LTS machine with an Intel Xeon E5-2660 CPU and 64 GiB of RAM. We ran each benchmark program on its reference inputs and measured the average run-time over three runs. Figure 3.2 shows the results of these tests. We compiled each benchmark with a vanilla clang/LLVM 3.9.1 compiler and optimization level `-O3` to establish a baseline. We then compiled the benchmarks with our modified clang/LLVM 3.9.1 compiler to generate the HexVASAN results.

The geometric mean overhead in these benchmarks was just 0.45%, indistinguishable from measurement noise. The only individual benchmark result that stands out is that of `libquantum`. This benchmark program performed 880M variadic function calls in a run of just 433 seconds.

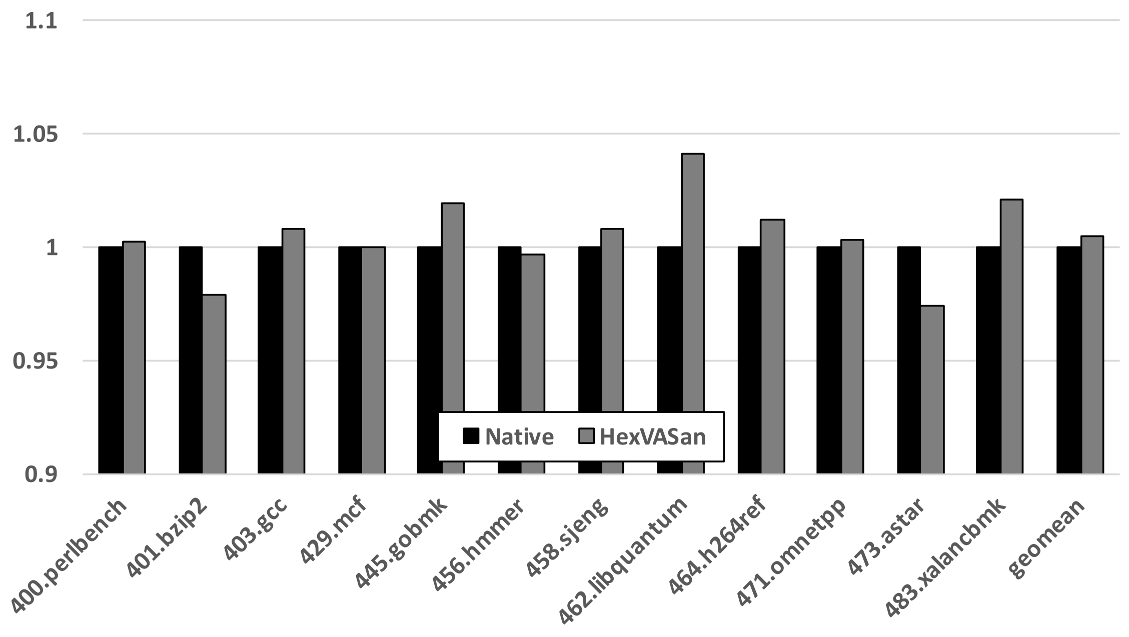


Figure 3.2. Run-time overhead of HexVASAN in the SPECint CPU2006 benchmarks, compared to baseline LLVM 3.9.1 performance.

3.7 Related work

HexVASAN can either be used as an always-on runtime monitor to mitigate exploits or as a sanitizer to detect bugs, sharing similarities with the sanitizers that exist primarily in the LLVM compiler. Similar to HexVASAN, these sanitizers embed run-time checks into a program by instrumenting potentially dangerous program instructions.

AddressSanitizer [34] (ASan), instruments memory accesses and allocation sites to detect spatial memory errors, such as out-of-bounds accesses, as well as temporal memory errors, such as use-after-free bugs. Undefined Behavior Sanitizer [36] (UBSan) instruments various types of instructions to detect operations whose semantics are not strictly defined by the C and C++ standards, e.g., increments that cause signed integers to overflow, or null-pointer dereferences. Thread Sanitizer [82] (TSAN) instruments memory accesses and atomic operations to detect data races, deadlocks, and various misuses of synchronization primitives. Memory Sanitizer [35] (MSAN) detects uses of uninitialized memory.

CaVer [83] is a sanitizer targeted at verifying correctness of downcasts in C++. Downcasting converts a base class pointer to a derived class pointer. This operation may be unsafe as it cannot be statically determined, in general, if the pointed-to object is of the derived class type. TypeSan [84] is a refinement of CaVer that reduces overhead and improves the sanitizer coverage.

UniSan [85] sanitizes information leaks from the kernel. It ensures that data is initialized before leaving the kernel, preventing reads of uninitialized memory.

All of these sanitizers are highly effective at finding specific types of bugs, but, unlike HexVASAN, they do not address misuses of variadic functions. The aforementioned sanitizers also differ from HexVASAN in that they typically incur significant run-time and memory overhead.

Different control-flow hijacking mitigations offer partial protection against variadic function attacks by preventing adversaries from calling variadic functions through control-flow edges that do not appear in legitimate executions of the program. Among these mitigations, we find Code Pointer Integrity (CPI) [86], a mitigation that prevents attackers from overwriting code pointers in the program, and various implementations of Control-Flow Integrity (CFI), a technique that does not prevent code pointer overwrites, but rather verifies the integrity of control-flow transfers in the program [6, 23, 26, 79, 80, 87–108].

Control-flow hijacking mitigations *cannot* prevent attackers from overwriting variadic arguments directly. At best, they can prevent variadic functions from being called through control-flow edges that do not appear in legitimate executions of the program. We therefore argue that HexVASAN and these mitigations are orthogonal. Moreover, prior research has shown that many of the aforementioned implementations fail to fully prevent control-flow hijacking as they are too imprecise [9, 58, 109, 110], too limited in scope [59,111], vulnerable to information leakage attacks [112], or vulnerable to spraying attacks [113,114]. We further showed in Section 3.6.1 that variadic functions exacerbate CFI's imprecision problems, allowing additional leeway for adversaries to attack variadic functions.

Defenses that protect against direct overwrites or misuse of variadic arguments have thus far only focused on format string attacks, which are a subset of the possible attacks on variadic functions. LibSafe detects potentially dangerous calls to known format string functions such as `printf` and `sprintf` [115]. A call is considered dangerous if a `%n` specifier is used to overwrite the frame pointer or return address, or if the argument list for the `printf` function is not contained within a single stack frame. FormatGuard [116] instruments calls to `printf` and checks if the number of arguments passed to `printf` matches the number of format specifiers used in the format string.

Shankar et al. proposed to use static taint analysis to detect calls to format string functions where the format string originates from an untrustworthy source [117]. This approach was later refined by Chen and Wagner [118] and used to analyze thousands of packages in the Debian 3.1 Linux distribution. TaintCheck [119] also detects untrustworthy format strings, but relies on dynamic taint analysis to do so.

None of these solutions provide comprehensive protection against variadic argument overwrites or misuse.

3.8 Conclusions

Variadic functions introduce an implicitly defined contract between the caller and callee. When the programmer fails to enforce this contract correctly, the violation leads to runtime crashes or opens up a vulnerability to an attacker. Current tools, including static type checkers and CFI implementations, do not find variadic function type errors or prevent attackers from exploiting calls to variadic functions. Unfortunately, variadic functions are prevalent. Programs such as SPEC CPU2006, Firefox, Apache, CPython, nginx, wireshark and libraries frequently leverage variadic functions to offer flexibility and abundantly call these functions.

We have designed a sanitizer, HexVASAN, that addresses this attack vector. HexVASAN is a light weight runtime monitor that detects bugs in variadic functions and prevents the bugs from being exploited. It imposes negligible overhead (0.45%) on the SPEC CPU2006 benchmarks and is effective at detecting type violations when calling variadic arguments. This part of the thesis has been published in the 26th USENIX Security Symposium, 2017 [120].

	Call sites		Func.			Ratio	
Program	Tot.	Ind.	Tot.	A.T.	Proto	Tot.	A.T.
Firefox	30225	1664	421	18	241	1.75	0.07
Chromium	83792	1728	794	44	396	2.01	0.11
FreeBSD	189908	7508	1368	197	367	3.73	0.53
Apache	7121	0	94	29	41	2.29	0.71
CPython	4183	0	382	0	38	10.05	0.00
nginx	1085	0	26	0	14	1.86	0.00
OpenSSL	4072	1	23	0	15	1.53	0.00
Wireshark	37717	0	469	1	110	4.26	0.01
perlbench	1460	1	60	2	18	3.33	0.11
bzip2	85	0	3	0	3	1.00	0.00
gcc	3615	55	125	0	31	4.03	0.00
mcf	29	0	3	0	3	1.00	0.00
milc	424	0	21	0	8	2.63	0.00
namd	485	0	24	2	8	3.00	0.25
gobmk	2911	0	35	0	8	4.38	0.00
soplex	6	0	2	1	2	1.00	0.50
povray	1042	40	45	10	16	2.81	0.63
hmmer	671	7	9	1	5	1.80	0.20
sjeng	253	0	4	0	3	1.33	0.00
libquantum	74	0	91	0	7	13.00	0.00
h264ref	432	0	85	5	13	6.54	0.38
lbm	11	0	3	0	2	1.50	0.00
omnetpp	340	0	48	23	19	2.53	1.21
astar	42	0	4	1	4	1.00	0.25
sphinx3	731	0	20	0	5	4.00	0.00
xalancbmk	19	0	4	2	4	1.00	0.50

Table 3.2.
Statistics of Variadic Functions for Different Benchmarks. The second and third columns are variadic call sites broken into "Tot." (total) and "Ind." (indirect). The third and fourth columns are for variadic functions. "A.T." stands for *address taken*. "Proto." is the number of distinct variadic function prototypes. "Ratio" indicates the *function-per-prototypes* ratio for variadic functions.

Benchmark		Native		HexVASAN	
Octane	AVERAGE	31241	80	30907	73
	STDDEV	2449	82	2442	82
	OVERHEAD			-1	08%
JetStream	AVERAGE	200	76	198	75
	STDDEV	0	66	1	68
	OVERHEAD			-1	01%
Kraken	AVERAGE [ms]	832	48	832	41
	STDDEV [ms]	7	41	12	71
	OVERHEAD			0	01%

Table 3.3.
Performance overhead on Firefox benchmarks. For Octane and JetStream higher is better, while for Kraken lower is better.

4 ARTEMIS

4.1 Motivation

Cryptography is instrumental in securing our electronic communications and systems; yet time and time again they are mis-used, mis-implemented, or created in an ad-hoc manner. Additionally, while cryptography plays a fundamental role in securing systems, it is unfortunately also often used for malicious purposes. It has been seen in practice that cryptographic functions are extensively used from hiding payloads in malware to bypassing security protocols. In both situations (misuse and malicious use) many of these instances occur in closed-source code or binary applications, which inherently present a challenge for independent audit and analysis. Therefore, detecting the presence of cryptographic functions in a binary application can be both a pragmatic indicator of malicious behaviors as well as a point of interest to understand the effectiveness of dynamic analysis.

Motivated by these insights, as well as others, a variety of work has been performed across industry and academia to develop techniques and build tools that can identify different cryptographic primitives in binary applications. In this chapter, we provide a systematic study and comparison of these approaches by developing a standard framework with a focus on their effectiveness in identification in many different scenarios. In particular, we build a taxonomy of the current available detection criteria and highlight the strengths and weaknesses of each, as well as provide a mapping between detection criteria and specific types or classes of cryptographic algorithms. Additionally, we highlight major gaps in existing work, especially as they relate to modern cryptographic primitives and real-world use cases, and discuss a variety of avenues for the future.

4.2 Introduction

Cryptographic functions play a key role in securing communications and are imperative in modern systems and applications. From data integrity to authentication and online banking to messaging friends, cryptography is everywhere. Well-known cryptographic libraries, such as OpenSSL [121], are widely used not only to generate TLS certificates and validate certificate information, but also to implement cryptography in general purpose applications. Despite their universal benign usage, cryptographic functions are also used in malware and to evade security protocols. In recent ransomware attacks [122, 123], cryptographic functions have been used to encrypt a victim's information and later asked to pay ransom in exchange of recovering their files and information.

On one hand, the usage of cryptographic functions makes it easier to carry out secure operations, and on the other hand, its malicious usage by attackers makes it harder for cryptography and security experts to perform forensic and reverse engineering. Hence, the ability to automatically identify or detect the presence of cryptographic functions in binary applications can help these experts in a number of ways. First, it can assist in binary analysis to give a better depiction of how the functions work. Second, by determining the type of cryptographic function in a binary can help to pinpoint the existence of a malicious payload [124]. Finally, oftentimes malware and ransomware uses = similar cryptographic functions with some minor changes. For example, there are different versions of the TeslaCrypt ransomware [125]. Identifying one ransomware via an automatic detection mechanism can help to identify other versions and thus save time and resources to could be better devoted to implementing effective prevention mechanisms.

There are many dimensions of identifying cryptography in binaries, e.g., finding keys (based on different encoding formats), identifying copied functions, or identifying reimplementations of existing cryptographic algorithms. Each aspect has its own set of challenges. Finding keys in an obfuscated binary can be challenging due to different

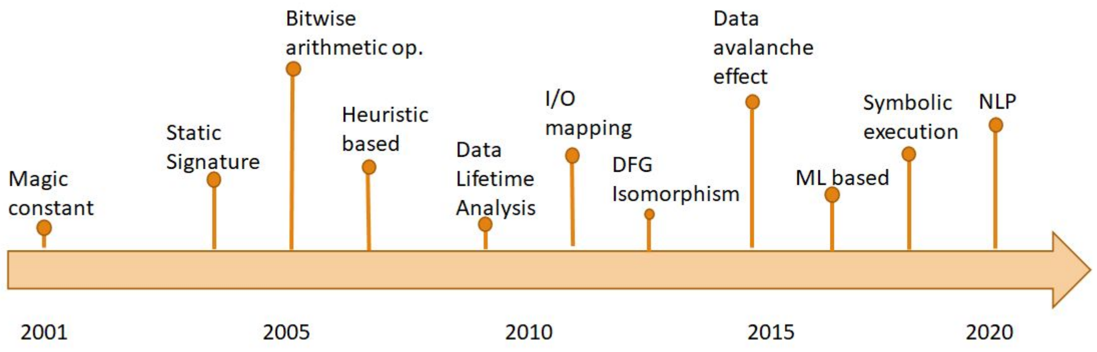


Figure 4.1. Evolution of the research techniques to identify cryptographic functions over time

encodings and formats. Copied functions can be identified based on function behaviors or I/O mappings. Finally, it is extremely difficult to identify implementations of certain cryptographic algorithms, since there is no gold standard for implementation and even simple obfuscations can prohibit the identification of an algorithm. This study primarily focuses on identifying cryptographic algorithms, the key challenge when analyzing unknown code, and Section 4.6 discusses more of the challenges of identifying cryptographic algorithms.

However, cryptographic function identification has its own sets of challenges. Binaries can be obfuscated in a variety of ways, such as control-flow flattening, data-splitting, data-aggregation, inclusion of bogus control-flow. In addition, simple cryptographic algorithms can be implemented in different ways, hence detection mechanism that may work on one variation, may fail on its complicated/ simplified implementations. Malware authors also generates packed binary to evade any possible detection (to make it harder for detection).

Cryptographic functions tend to have a lot of mathematical computations, nested loop operations, exclusive input-output mapping which are distinct from non-cryptographic functions. Researchers use these features as the medium of identification. Figure 4.1 shows the timeline of the research of cryptographic function identification approaches.

Harvey et al. [126] first utilized magic constants to identify cryptographic primitives in binary in 2001 as hash functions tend to use one or more specific constant values. Later works focused on signature based detection mechanisms. However, due to the limitations of signature based approaches, subsequent work placed a greater emphasis on heuristic based detection such as identification of particular basic blocks, instruction patterns etc. With the popularization of modern machine learning algorithms, newer works utilized deep learning and AI in general to extract cryptographic features from a binary. We discuss these various approaches in depth in Section 4.4.

The primary contributions of **Artemis** are:

- We present a systematic study of cryptographic function identification approaches.
- We create a standardized suite of performance metrics and benchmarks to evaluate the effectiveness of current detection mechanisms and analyze existing tools based on this suite.
- Based off of this analysis, we discuss the research gaps in this domain and propose directions for future work.
- We present a comprehensive framework to understand the scalability and impact of dynamic analysis in detection mechanisms.

4.3 Research gap in identification of cryptographic algorithms

Identification approaches can differ substantially based on the features of the algorithm. Hence, it is crucial to understand the key differences and major features of the algorithms. First, we present the classification of cryptographic algorithms.

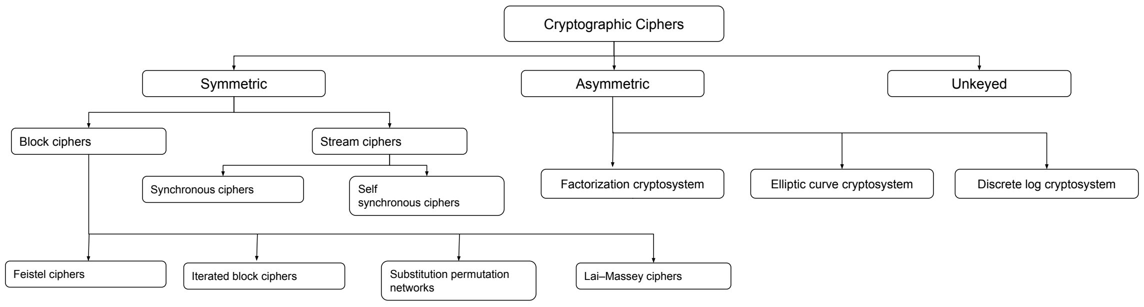
Based on the usage of keys [127], cryptographic algorithms can be broadly divided into three categories: i. Symmetric key, ii. Asymmetric key, and iii. Unkeyed (Hashing). Symmetric key algorithms use a secret key for both encryption and decryption. Asymmetric key cryptography also known as public-key algorithms use paired keys usually consisting of a public and a private key. Generally, the public key is known to all and the private key is kept secret by the owner. Unlike the first two types, hashing refers to encryption without using any key. 4.3 shows the high level taxonomy of the algorithms and 4.2(b) shows the number of different categories of algorithms that have been used in cryptographic function identification approaches. We can see that, most of the detection approaches are primarily focused on substitution-permutation network based algorithms such as AES. There are some works on Feistel network based algorithm like DES and factorization based approaches like RSA. However, we can see that there has not been any work done on elliptic curve cryptography as well as discrete log based cryptographic algorithms despite their wide usage in modern systems and applications. Hence, we believe it is important that future researches in this domain consider such algorithms and come up with newer detection mechanisms.

4.4 Cryptographic Features

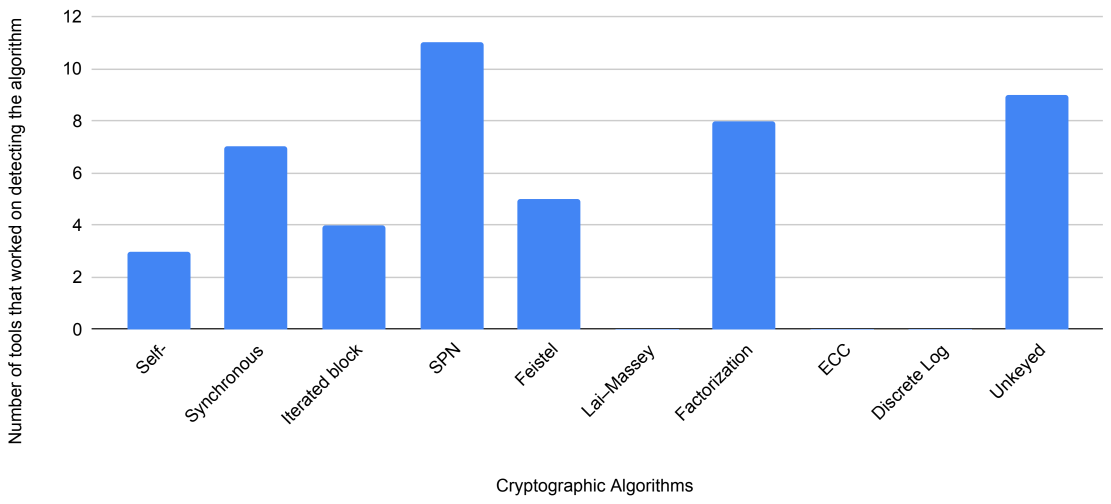
Cryptographic functions have their own set of characteristics, and researchers have used these characteristics in identifying such functions. Some of these features only pertain to certain cryptographic algorithms and cannot work in identifying other cases. In the following subsections, we discuss some of the popular cryptographic features used for identification purposes.

4.4.1 Magic Constants

In this approach, a tool or individual searches for algorithm-specific magic constants. For example, the magic constants for TEA algorithms usually are 2654435769 or 0x9E3779B9. Existence of these constants in a binary may indicate the presence of the TEA algorithm.



(a) Taxonomy of Cryptographic Algorithms



(b) Number of specific algorithms used in different cryptographic function identification approaches

4.4.2 Presence of Loops

Most cryptographic functions use some form of loops for key generation or for encryption/decryption purposes. For example, the factorization-based algorithm RSA has loops to perform modular exponentiation.

4.4.3 Changes in Entropy

The decryption procedure in cryptographic functions tends to reduce information entropy. Researchers have used taint analysis to check if a memory buffer is decrypted by measuring its entropy.

4.4.4 I/O Mapping

Cryptographic functions, such as decryption, have a one-to-one mapping from their input to output, which means given a key and the plaintext, one would always get the same output. However, these approaches rely on the accurate extraction of key and input from the memory. Any wrong extraction can defeat the purpose of unique mapping.

4.4.5 Data-Flow Isomorphism

A data-Flow graph (DFG) [128] is a graph that shows data dependencies between different operations. In this approach, a DFG is being generated from the corresponding assembly code, and later on subgraphs in the DFG are checked to see whether they are isomorphic to the graph signature of a particular cryptographic algorithm. It was first proposed by Lestringant et al. [129] and primarily used to identify symmetric key based algorithms. However, this approach is limited in case of conditional statements which are more prominent in asymmetric algorithms. Secondly, DFG can vary depending on the data obfuscations techniques such as data-splitting.

4.4.6 Instruction Sequence

Cryptographic algorithms execute sequences of instructions for encryption and decryption purposes. Detection tools take advantage of these sequences to generate a static signature of the algorithm. However, any form of control-flow obfuscation can change the execution of instruction sequences.

Overall, research on cryptographic function identification is sliding from static based features to more dynamic oriented features. Looking for particular sequences can be easily defeated by various form of obfuscations and compiler level optimizations. However, dynamic feature based approaches like flow analysis are more resilient to such techniques.

4.5 Categorization of detection approaches

We have broadly categorized the detection mechanisms into three categories: i) Dynamic, ii) Static, and iii) Machine learning based approaches.

4.5.1 Static Approaches

Static approaches rely on static signatures, for e.g., 'magic' constants, instruction sequences as well as different code structures such as S-box to identify cryptographic algorithms. Harvey et al. [126] first proposed [130] identification of cryptographic algorithms in binary files in a research report. The report focused on finding algorithms based on their constant characteristics. By taking advangtage of feature matching, subsequent works [131–133] primarily focused on protocol reverse engineering. Lestringant et al [129] used data-flow iso-morphism to find symmetric key algorithms.

Commonly used tools such as Draft Crypto Analyzer (DRACA) [134], Kanal [135], Kerckhoffs [136], Hash & Crypto Detector (HCD) [137], Signsrch [138], Crypto Searcher [139], Findcrypt [140] and IDAscope, utilize static signature patterns of different cryptographic functions. Static based approaches do not have any performance overhead. However, these detection mechanisms can be easily bypassed using simple obfuscation techniques.

4.5.2 Dynamic Approaches

Dynamic approaches [141–143] primarily focus on identifying cryptographic primitives from execution traces. Lutz et al. [144] first applied dynamic analysis in execution traces to identify cryptographic algorithms based on three indicators: i) presence of loops, ii) changes in entropy, and iii) high ratio of bitwise arithmetic instructions. Based on the data avalanche effect, CipherXRay [143] pinpoints the boundary of cryptographic operations and recovers transient cryptographic secrets. However, this approach does not work in case of stream ciphers as they don't show any data avalanche effect.

Gröbert et al. [145] proposed heuristic based approaches on both generic characteristics of cryptographic code and on signatures for specific instances of cryptographic algorithms by mapping input-output (I/O) relations. Aligot [142] further extends this idea of I/O mapping in cryptographic function identification. It retrieves I/O parameters in an implementation-independent fashion, and compares them with known cryptographic functions as well as performs an inter-loop data flow analysis. CryptoHunt [141] uses bit-precise symbolic loop mapping to identify cryptographic functions and applies guided fuzzing to make the solution scalable. Dynamic approaches, in general, perform better than static approaches for obfuscated binaries, but suffer from performance overhead.

4.5.3 Machine Learning Based Approaches

With the advent of machine learning techniques, several researches have used such techniques in identification of cryptographic algorithms. Falke [146] proposed a neural network based approach by modeling classifiers for arbitrary cryptographic algorithms from sample files and then automatically extracting features to train the neural network. It offered a high detection rate in combination with a low false positive rate. Benedetti et al. [147] used 'grap' tool to detect cryptographic algorithms by creating patterns for AES and ChaCha20. Jia et al. [130] proposed a NLP based approach which first extracts the semantic information of assembly instructions. It then transfers them into 100-dimensional vectors and later uses K-Max-CNN-Attention to classify cryptographic functions. Hill et al. [148] proposed a Dynamic Convolutional Neural Network based learning system CryptoKnight which learns from new cryptographic execution patterns to classify unknown software.

4.6 Challenges

Identifying cryptographic functions has its unique set of challenges. Here, we discuss the key challenges.

4.6.1 Obfuscation

The initial purpose of obfuscation was to protect software intellectual property [149] from malicious reverse engineering attempts. However, malware authors adopted obfuscation techniques as a way to avoid detection. Various types of obfuscation techniques have been implemented to thwart any form of detection. The obfuscation techniques can be primarily categorized as follows:

- Control-flow obfuscation
- Data obfuscation
- Layout obfuscation

To hinder, any form of CFG based detection, malware authors introduce various techniques such as inclusion of bogus control-flow, control-flow flattening [150], and opaque predicate [151]. Similar to control flow obfuscation, data obfuscation techniques, such as data aggregation and data splitting, attempt to hinder any detection approaches based on input/output relationships. Listing 4.2 shows an example of data obfuscation. Here, the variable *var1* used in Listing 4.1 can be split into two variables *var11* and *var12* to avoid detection, although both of the programs are performing the similar operations in the end.

```
int var1 = func1();
int var2 = func2();

while(condition statement) {

    int m = var1 << 4;
    int n = var2 *5;

}
```

Listing 4.1 Normal program

```
int var1 = func1();
int var2 = func2();

short var11 = var1 & 000fffff;
short var12 = var1 >> 20 & 00000fff;
while(condition statement) {

int new_var = (int) var12 << 20 | var11;

m = new_var << 4;
n = var2 * 5;
}
```

Listing 4.2 Data splitting

For layout obfuscation, malware authors perform address obfuscation and debug information obfuscation, as well as address layout/memory layout randomization.

4.6.2 Implementation Variation

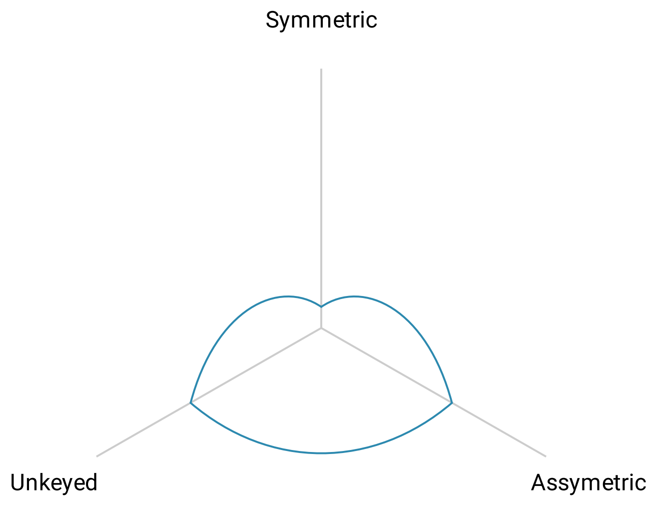
One cryptographic algorithm can be implemented in a number of ways. For example, in malware Storm Worm and Silent Banker, researchers discovered a buggy implementation of TEA algorithm [142]. Hence, even if a detection approach can detect an ideal implementation of cryptographic algorithms, there is no guarantee that the approach can also detect all the implementation variations of the same algorithm.

4.6.3 Differences in Cryptographic Functions

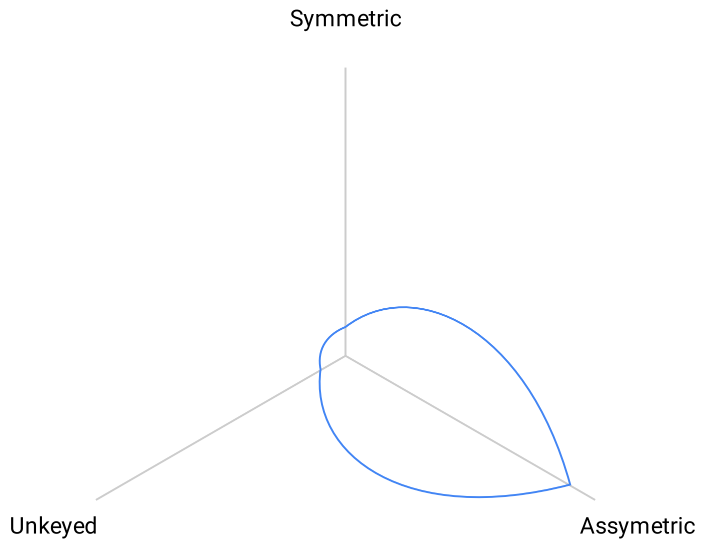
There are fundamental differences in different cryptographic algorithms. Different algorithms use different set of features for identification purposes. For example, the data avalanche effect [143] which is based on the fact that an insignificant change in the input parameters can make significant differences in the output, works for block ciphers. In case of stream ciphers, there are no such observations like data-avalanche.

4.7 Performance Metric

We propose a performance metric to conduct a fair comparison of all the available tools. 4.7 shows the score for each of the categories. We have considered a comprehensive set of obfuscation and optimization flags to mimic real world scenarios. For each true positive and true negative identification, we assign a score of 1. To calculate the final score, we take the average of true positive and true negative scores. Our analysis focuses on precision for each of this tool along three dimensions (symmetric, asymmetric and unkeyed cryptography). For example, if tool **A** can identify MD5 with all 6 optimization flags both for true positive and true negative cases, but fail to detect with any form of obfuscation, then based on our metric, we give tool **A**



(c) Evaluation of DRACA based on performance metrics



(d) Evaluation of Signsrch based on performance metrics

Metric	Score
Failed Identification	0
Identification with no optimization/obfuscation	1
Identification with optimization flag -O0	1
Identification with optimization flag -O1	1
Identification with optimization flag -O2	1
Identification with optimization flag -O3	1
Identification with optimization flag –Os	1
Identification with optimization flag: -Ofast	1
Identification with instruction substitution obfuscation: -obs_sub	1
Identification with control-flow flattening: -obs_fla	1
Identification with bogus control-flow: -obs_ bcf	1
Identification with combination of obfuscation: -obs_sub, obs_fla, obs_bcf	1
Identification with combination of obfuscation and optimization: -obs_sub, obs_fla, obs_bcf, -O3	1

Table 4.1.
Score for each of the evaluation criterion based on different optimization and obfuscation flags.

score of 7 in unkeyed category. We believe this metric gives us better insight on the effectiveness of detection approaches. 4.2(c) and 4.2(d) shows the performance of the two tools namely DRACA and Signsrch based on the above mentioned metrics. We plan to work on other tools in future.

4.8 Benchmarks

We believe that to do a fair evaluation in any domain requires a standard benchmark. We propose a benchmark that will help us understand the scalability and effectiveness of a detection approach. We have three categories in our benchamrk: i) microbenchmarks, ii) libraries, and iii) large project. We believe that to understand the scalability of a mechanism, it it crucial to determine that the tool performs well irrespective of the code size and its applications. Our microbenchmarks category contains small programs on file manipulation, networking, I/O heavy program, math heavy program and golden implementation of well-known cryptographic algorithms. We have chosen small programs such as I/O heavy and math-heavy programs since they have the similar instruction sets or behavior patterns like cryptographic functions. For libraries, we have chosen a set of total eleven cryptographic libraries as well as encoding and compression libraries that have similar functionalities like cryptographic functions.

- openssl
- libgcrypt
- libsodium
- mbedTLS
- gnuTLS
- bzip2
- zlib
- ffmpeg
- libgsm
- libjpeg
- libpng

For large scale project, we have chosen signal given its large code base and wide spread usage. Table 4.2 shows the analysis of two tools DRACA and signsrch across all the three categories of the benchmarks as well as with different compilation and obfuscation flags.

Table 4.2.
Analysis of the tools across the three categories of the benchamark

Tools		Micro-benchmarks				Libraries											Large Scale Project
		File	I/O	Maths	Network	openssl	libgcrypt	libsodium	mbedTLS	gnuTLS	bzip2	zlib	ffmpeg	libgsm	libjpeg	libpng	signal
DRACA	None	✓	✓	✓	✓	✓	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗
	-O0	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-O1	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-O2	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-O3	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-Os	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-Ofast	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	obs_sub	✓	✓	✓		✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	obs_fla	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	obs_bcf	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-obs_sub, obs_fla, obs_bcf	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-obs_sub, obs_fla, obs_bcf, -O3	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Signsrch	None	✓	✓	✓	✓	✗	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗
	-O0	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-O1	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-O2	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-O3	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-Os	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-Ofast	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	obs_sub	✓	✓	✓		✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	obs_fla	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	obs_bcf	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-obs_sub, obs_fla, obs_bcf	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	-obs_sub, obs_fla, obs_bcf, -O3	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

4.9 Case study: Openssl

We have analyzed the performance of the tools on openssl. We have observed that static tools such as Draca, Findcrypt2, Signsrch, CryptoSearcher, Hash & Crypto detector can detect only a handful of algorithms from the pool of RSA, AES, DES, TEA and RC4. None of the tools succeed to truly identify all the algorithms. This scenario becomes even worse on the presence of obfuscation. However, dynamic based approaches such as Aligot are able to identify these algorithms. This helps us to draw the conclusion that dynamic analysis based approaches can perform better identification than static ones.

4.10 Conclusion and Future Work

Artemis presents a study on cryptographic function detection approaches. We have presented results for some of the existing tools. In the future we plan to provide results for all available tools. The list of such tools are:

- DRACA
- IDA Scope
- Hash Crypto Detector
- Signsrch
- SnD Crypto Scanner
- Findcrypt2
- KANAL
- Kerckhoff
- CryptoHunt
- CryptoKnight

- Aligot
- Where's the Crypto

Once we reproduce the results of all selected tools, we hope to have a more complete picture of their detection capabilities and potential failure classes. We are particularly interested in biases towards specific algorithms or classes of algorithms. From our initial result, we strongly believe the benchmark will establish a standard for future evaluation of tools. The performance metrics will set a fair comparison ground between different tools. By doing so, **Artemis** creates an avenue for future research in this domain.

5 SUMMARY

State-of-the-art CFI approaches are over-approximate due to the static nature of the analyses and leave several areas unprotected such as variadic functions and code pointers. We strengthen CFI along these two unprotected dimensions by providing tighter enforcement mechanism using dynamic analysis and then analyze its applications on real-world programs.

We have proposed three novel techniques in this report, namely **Ancile**, **HexVASAN**, and **FitJit**, as well as we provide a systematic study (**Artemis**) on the effectiveness of dynamic analysis in real world programs. Among the three mechanisms, we have implemented and evaluated **Ancile** and **HexVASAN** to secure code-pointers and variadic functions. We have investigated real-world programs in **Artemis** and as a next step, we plan to extend CFI across language boundaries by implementing **FitJit**.

Ancile To overcome over-approximation of static CFI policies, we have developed **Ancile** which enforces strict target sets on indirect control-flow transfers. It uses seed demonstrated fuzzing for target discovery and can reduce the target sets up to 97.4%. It is also evident from our case studies that **Ancile** makes sensitive functions more difficult for attackers to reach, essentially raising the bar for CFH attacks.

HexVASAN To defend against CFH attacks via variadic functions, we have implemented **HexVASAN**, a light-weight runtime monitor to detect bugs in variadic functions with a negligible performance overhead (0.45%).

Artemis To show the applications of dynamic analysis, we present **Artemis**. We propose performance metrics to compare the efficacy of different tools in regarding different optimization, and obfuscation techniques. We also developed a framework to understand the scalability and the application of each tool.

FitJit To secure language boundaries against CFH attacks, we propose **FitJit** to enforce type and control-flow integrity in JIT compiler and jit-compiled code. In the future, we plan to work on the implementation of **FitJit** according to the design policy proposed in Section 2.10.

Outlook Our techniques show that the potential of dynamic analysis applied to source code and binary code. We leverage dynamic analysis to extract precise information of the underlying system, showing that these analyses play a paramount role in securing C/C++ against different attack vectors. Additionally, we show that dynamic analysis exhibits great promise in other domains of binary analysis, such as helping to identify cryptographic code in binaries.

REFERENCES

REFERENCES

[1] PaX Team, "Pax address space layout randomization (aslr)," 2003.

[2] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security Symposium*, 1998.

[3] PaX Team, "PaX non-executable pages design & implementation," http://pax.grsecurity.net/docs/noexec.txt, 2004.

[4] A. Sadeghi, S. Niksefat, and M. Rostamipour, "Pure-call oriented programming (pcop): chaining the gadgets using call instructions," *Journal of Computer Virology and Hacking Techniques*, pp. 1–18, 2017.

[5] 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.* ACM, 2007, pp. 552–561.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security.* ACM, 2005, pp. 340–353.

[7] "Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd," https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/, Online; accssed 13-Oct-2020.

[8] "Linux kernel," https://en.wikipedia.org/wiki/Linux_kernel, Online; accssed 13-Oct-2020.

[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity." in *USENIX Security Symposium*, 2015, pp. 161–176.

[10] 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 *Security and Privacy (SP), 2016 IEEE Symposium on.* IEEE, 2016, pp. 969–986.

[11] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2018, pp. 1868–1882.

[12] "Control flow integrity design documentation," https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html.

[13] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "{RAZOR}: A framework for post-deployment software debloating," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1733–1750.

[14] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 380–394.

[15] A. Quach, A. Prakash, and L. K. Yan, "Debloating software through piece-wise compilation and loading," *arXiv preprint arXiv:1802.00759*, 2018.

[16] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 476–486.

[17] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 70–83.

[18] M. D. Brown and S. Pande, "Is less really more? towards better metrics for measuring security improvements realized through software debloating," in *12th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 19)*, 2019.

[19] D. Kim, W. N. Sumner, X. Zhang, D. Xu, and H. Agrawal, "Reuse-oriented reverse engineering of functional components from x86 binaries," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1128–1139. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568296

[20] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu, "Reddroid: Android application redundancy customization based on static analysis," in *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering (ISSRE'18)*, 2018.

[21] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: Size does matter in turing-complete return-oriented programming," in *WOOT*, 2012.

[22] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys(CSUR)*, vol. 50, no. 1, p. 16, 2017.

[23] 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 *USENIX Security Symposium*, 2014, pp. 941–955.

[24] R. Gawlik and T. Holz, "Towards automated integrity protection of c++ virtual function tables in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 396–405.

[25] X. Fan, Y. Sui, X. Liao, and J. Xue, "Boosting the precision of virtual call integrity protection with partial pointer analysis for c++," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.

[26] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2015, pp. 914–926.

[27] 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.* ACM, 2015, pp. 927–940.

[28] 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 ({USENIX} Security 17)*, 2017, pp. 131–148.

[29] 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.* ACM, 2018, pp. 1470–1486.

[30] "Fuzzing," https://www.owasp.org/index.php/Fuzzing.

[31] "american fuzzy lop," http://lcamtuf.coredump.cx/afl/.

[32] "honggfuzz," https://github.com/google/honggfuzz, Online; accssed 11-Oct-2020.

[33] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," *arXiv preprint arXiv:1806.04355*, 2018.

[34] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[35] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization.* IEEE Computer Society, 2015, pp. 46–55.

[36] "Undefinedbehaviorsanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[37] M. Corporation, "A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003," https://support.microsoft.com/en-us/kb/875352, 2013.

[38] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *SEC '98*, 1998.

[39] "New memory corruption attacks:why can't we have nice things?" http://nebelwelt.net/publications/files/1532c3-presentation.pdf.

[40] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation.* Peter Sestoft, 1993.

[41] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[42] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[43] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[44] M. Ghaffarinia and K. W. Hamlen, "Binary control-flow trimming," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1009–1022.

[45] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," *SP'19*, 2019.

[46] "Addresssanitizer," https://github.com/google/sanitizers/wiki/AddressSanitizer.

[47] "libpng," http://www.libpng.org/pub/png/libpng.html.

[48] "libtiff," http://www.libtiff.org/.

[49] G. Android, "Kernel control flow integrity," https://source.android.com/devices/tech/debug/kcfi, 2018.

[50] G. Chromium, "Chromium: Control flow integrity," https://www.chromium.org/developers/testing/control-flow-integrity, 2017.

[51] "Control-flow enforcement technology," https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, Online; accssed 11-Oct-2020.

[52] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 585–598.

[53] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 329–339.

[54] N. Redini, R. Wang, A. Machiry, Y. Shoshitaishvili, G. Vigna, and C. Kruegel, "B in t rimmer: Towards static binary debloating through abstract interpretation," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 482–501.

[55] Y. Jiang, C. Zhang, D. Wu, and P. Liu, "Feature-based software customization: Preliminary analysis, formalization, and methods," in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 2016, pp. 122–131.

[56] Y. Jiang, D. Wu, and P. Liu, “Jred: Program customization and bloatware mitigation based on static analysis,” in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 12–21.

[57] “Control flow integrity,” https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[58] 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*. ACM, 2015, pp. 901–913.

[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 *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 745–762.

[60] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, “Finding and preventing bugs in javascript bindings,” in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 559–578.

[61] “Issue 1053604: Security: Incorrect side effect modelling for jscreate,” https://bugs.chromium.org/p/chromium/issues/detail?id=1053604, Online; accssed 13-Oct-2020.

[62] R. Gawlik and T. Holz, “Sok: Make jit-spray great again,” in *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[63] B. Niu and G. Tan, “Rockjit: Securing just-in-time compilation using modular control-flow integrity,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1317–1328.

[64] 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 *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 355–366.

[65] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, “Librando: transparent code randomization for just-in-time compilers,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 993–1004.

[66] T. Wei, T. Wang, L. Duan, and J. Luo, “Insert: Protect dynamic code generation against spraying,” in *International Conference on Information Science and Technology*. IEEE, 2011, pp. 323–328.

[67] R. Wu, P. Chen, B. Mao, and L. Xie, “Rim: A method to defend from jit spraying attack,” in *2012 Seventh International Conference on Availability, Reliability and Security*. IEEE, 2012, pp. 143–148.

[68] P. Chen, R. Wu, and B. Mao, “Jitsafe: a framework against just-in-time spraying attacks,” *IET Information Security*, vol. 7, no. 4, pp. 283–292, 2013.

[69] P. Chen, Y. Fang, B. Mao, and L. Xie, “Jitdefender: A defense against jit spraying attacks,” in *IFIP International Information Security Conference.* Springer, 2011, pp. 142–153.

[70] Linux Programmer’s Manual, “va_start (3) - Linux Manual Page.”

[71] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, “System v application binary interface,” *AMD64 Architecture Processor Supplement, Draft v0.99*, 2013.

[72] “Cve-smbclient,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1886.

[73] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8617.

[74] “Fortify source,” https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html.

[75] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, “Fast byte-granularity software fault isolation,” in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[76] “Information technology – programming languages – c. standard,” International Organization for Standardization, Geneva, CH, Dec. 2011.

[77] “Information technology – programming languages – c++ standard,” International Organization for Standardization, Geneva, CH, Dec. 2014.

[78] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2004.

[79] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, “Ccfi: cryptographically enforced control flow integrity,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2015, pp. 941–951.

[80] Microsoft Corporation, “Control Flow Guard (Windows),” https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2016.

[81] “] exploit database. sudo debug privilege escalation,” https://www.exploit-db.com/exploits/25134.

[82] K. Serebryany and T. Iskhodzhanov, “Threadsanitizer: data race detection in practice,” in *Proceedings of the workshop on binary instrumentation and applications.* ACM, 2009, pp. 62–71.

[83] B. Lee, C. Song, T. Kim, and W. Lee, “Type casting verification: Stopping an emerging attack vector,” in *USENIX Security Symposium*, 2015.

[84] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, “Typesan: Practical type confusion detection,” in *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[85] K. Lu, C. Song, T. Kim, and W. Lee, “Unisan: Proactive kernel memory initialization to eliminate data leakages,” in *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[86] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity." in *OSDI*, vol. 14, 2014, p. 00000.

[87] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[88] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A framework to mitigate control-flow attacks on smartphones," in *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[89] B. Niu and G. Tan, "Monitor integrity protection with space efficiency and separate compilation," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[90] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *USENIX Security Symposium*, 2013.

[91] J. Pewny and T. Holz, "Control-flow restrictor: Compiler-based CFI for iOS," in *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[92] 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 *Security and Privacy (SP), 2013 IEEE Symposium on.* IEEE, 2013, pp. 559–573.

[93] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *USENIX Security Symposium*, 2013, pp. 337–352.

[94] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[95] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[96] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks," in *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[97] R. Gawlik and T. Holz, "Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs," in *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[98] 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 *Annual Design Automation Conference (DAC)*, 2014.

[99] 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.

[100] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz, "Opaque control-flow integrity," in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[101] A. Prakash, X. Hu, and H. Yin, "vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries," in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[102] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTint: Defending virtual function tables' integrity," in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[103] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "PathArmor: Practical ROP protection using context-sensitive CFI," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[104] P. Yuan, Q. Zeng, and X. Ding, "Hardware-assisted fine-grained code-reuse attack detection," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[105] D. Bounov, R. Kici, and S. Lerner, "Protecting C++ dynamic dispatch through vtable interleaving," in *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[106] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-Grained Control-Flow Integrity for Kernel Software," in *IEEE European Symp. on Security and Privacy*, 2016.

[107] B. Niu and G. Tan, "Modular control-flow integrity," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[108] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2015.

[109] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

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

[111] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[112] I. Evans, S. Fingeret, J. González, 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[113] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *USENIX Security Symposium*, 2016.

[114] E. Göktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining information hiding (and what to do about it)," in *USENIX Security Symposium*, 2016.

[115] T. Tsai and N. Singh, "Libsafe 2.0: Detection of format string vulnerability exploits," *white paper, Avaya Labs*, 2001.

[116] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "Formatguard: Automatic protection from printf format string vulnerabilities." in *USENIX Security Symposium*, 2001.

[117] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers." in *USENIX Security Symposium*, 2001.

[118] K. Chen and D. Wagner, "Large-scale analysis of format string vulnerabilities in debian linux," in *Proceedings of the 2007 workshop on Programming languages and analysis for security*, 2007.

[119] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Symposium on Network and Distributed System Security (NDSS)*, 2005.

[120] P. Biswas, A. Di Federico, S. A. Carr, P. Rajasekaran, S. Volckaert, Y. Na, M. Franz, and M. Payer, "Venerable variadic vulnerabilities vanquished," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 186–198.

[121] "Openssl," https://www.openssl.org/.

[122] "Wannacry ransomware," https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[123] "Petya ransomware," https://www.proofpoint.com/us/glossary/petya.

[124] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," *ieee seCurity & PrivaCy*, vol. 6, no. 5, pp. 65–69, 2008.

[125] "History and evolution of teslacrypt ransomware virus," https://www.engadget.com/2016-04-06-history-and-evolution-of-teslacrypt-ransomware-virus.html.

[126] I. Harvey, "Cipher hunting: How to find cryptographic algorithms in large binaries," *NCipher Corporation Ltd*, pp. 46–51, 2001.

[127] "Summary of cryptographic algorithms - according to nist," https://www.cryptomathic.com/news-events/blog/summary-of-cryptographic-algorithms-according-to-nist.

[128] "Data-flow graph," http://bears.ece.ucsb.edu/research-info/DP/dfg.html, Online; accssed 10-Dec-2020.

[129] P. Lestringant, F. Guihéry, and P.-A. Fouque, "Automated identification of cryptographic primitives in binary code with data flow graph isomorphism," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 203–214.

[130] L. Jia, A. Zhou, P. Jia, L. Liu, Y. Wang, and L. Liu, "A neural network-based approach for cryptographic function detection in malware," *IEEE Access*, vol. 8, pp. 23 506–23 521, 2020.

[131] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution." in *NDSS*, vol. 8, 2008, pp. 1–15.

[132] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 317–329.

[133] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," in *European Symposium on Research in Computer Security*. Springer, 2009, pp. 200–215.

[134] "Draft crypto analyzer," http://www.literatecode.com/draca.

[135] "Kanal - krypto analyzer for peid," http://www.dcs.fmph.uniba.sk/zri/6.prednaska/tools/PEiD/plugins/kanal.htm.

[136] "Kerckhoffs," https://github.com/felixgr/kerckhoffs.

[137] "Hash & crypto detector (hcd)," https://github.com/felixgr/kerckhoffs/blob/master/static_tools/HCD.rar.

[138] "Signsrch," http://aluigi.altervista.org/mytoolz/signsrch.zip.

[139] "Crypto searcher," http://x3chun.reteam.org/.

[140] "Findcrypt2," http://www.hexblog.com/?p=28.

[141] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 921–937.

[142] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: cryptographic function identification in obfuscated binary programs," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 169–182.

[143] X. Li, X. Wang, and W. Chang, "Cipherxray: Exposing cryptographic operations and transient secrets from monitored binary execution," *IEEE transactions on dependable and secure computing*, vol. 11, no. 2, pp. 101–114, 2012.

[144] N. Lutz, "Towards revealing attacker's intent by automatically decrypting network traffic," *Mémoire de maıtrise, ETH Zürich, Switzerland*, 2008.

[145] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 41–60.

[146] A. Aigner, “Falke-mc: A neural network based approach to locate cryptographic functions in machine code,” in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, 2018, pp. 1–8.

[147] L. Benedetti, A. Thierry, and J. Francq, “Detection of cryptographic algorithms with grap.” *IACR Cryptology ePrint Archive*, vol. 2017, p. 1119, 2017.

[148] G. Hill and X. Bellekens, “Cryptoknight: Generating and modelling compiled cryptographic primitives,” *Information*, vol. 9, no. 9, p. 231, 2018.

[149] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” 1997.

[150] C. Wang, J. Davidson, J. Hill, and J. Knight, “Protection of software-based survivability mechanisms,” in *2001 International Conference on Dependable Systems and Networks.* IEEE, 2001, pp. 193–202.

[151] C. Collberg, C. Thomborson, and D. Low, “Manufacturing cheap, resilient, and stealthy opaque constructs,” in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998, pp. 184–196.