NOVEL SYSTEM COMPARTMENTALIZATION AND REVERSE ENGINEERING METHODS

by

Derrick McKee

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

August 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Mathias Payer, Co-Chair

School of Computer Science

Dr. Sonia Fahmy, Co-Chair

School of Computer Science

Dr. Kihong Park

School of Computer Science

Dr. Dongyan Xu

School of Computer Science

Dr. Christina Garman

School of Computer Science

Approved by:

Dr. Kihong Park

To my parents, siblings, and teachers along my way.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Mathias Payer, for his guidance throughout my doctorate. Simply put, my success is due in large part to his unending help and encouragement to improve, and I am eternally grateful. I also thank my parents, siblings, and the rest of my family for their undying support and love. They mean the world to me, and I am lucky to have such a supportive, motivating group of people. I would like to thank my colleagues at MIT Lincoln Laboratory, particularly Dr. Nathan Burow and Dr. Hamed Okhravi. I have thoroughly enjoyed working with them, I have learned so much from them about being a good researcher, and I cannot wait to continue working with them for many years to come. Finally, I would like to thank the students and post-doctoral members of the HexHive lab at Purdue University and EPFL, and in particular Prashast Srivastava, Atri Bhattacharyya, Antony Vennard, and Hui Peng. They have provided valuable feedback, friendship, and much-needed frivolity.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

HAKC	Hardware-Assisted Kernel Compartmentalization
FlexC	Flexible Compartments
IOVFI	IOVec Function Identifier
IOVec	Input/Output Vector
CVE	Common Vulnerability Enumeration
IoT	Internet of Things
CTG	Call-and-Type Graph
ROP	Return-Oriented Programming
CIS	Characteristic IOVec Set
DCIS	Distinguishing Characteristic IOVec Set
FUT	Function Under Test

ABSTRACT

The need to secure software systems is more important than ever. However, while a lot of work exists to design and implement secure systems, a fundamental weakness remains. Instead of implementing software with least privilege policies, developers create monolithic systems that allow any instruction near universal memory access. This dissertation attempts to rectify this fundamental weakness to software design through three different contributions. First, I address the monolithic software design problem by proposing and evaluating a novel compartmentalization enforcement mechanism called Hardware-Assisted Kernel Compartmentalization (HAKC). HAKC is capable of enforcing an arbitrary compartmentalization policy using features of the ARMv9 ISA, without the need of any extra virtualization or trusted software layer. I then introduce a method of determining an optimal compartmentalization policy based on user performance and security constraints called FlexC, which is tested using HAKC as the enforcement mechanism. The end result is a hardened, compartmentalized kernel, customized to a user's needs, which enforces a least privilege policy that minimizes overhead. Finally, as an avenue for further compartmentalization policy generation, I introduce a novel program analysis framework called IOVec Function Identifier (IOVFI), which foregoes the use of language processing and model learning, but instead uses program state changes as a unique function fingerprint. I show that IOVFI is a more stable and accurate function identifier than the state-of-the-art, even in the presence of differing compilation environments, purposeful obfuscations, and even architecture changes.

1. INTRODUCTION

1.1 Problems with System Security and Semantic Identification

Secure software—software that is well tested, updated frequently with security patches, and utilizes the latest security mitigations, such as Address Space Layout Randomization, Data Execution Prevention, stack canaries, and Control-Flow Integrity—is more important today than ever before. We increasingly depend upon secure software, and serious calls are starting to be made for its importance to national security [1].

However, despite decades of research [2]–[4] into compiler-based defenses, run-time protections and immense computing resources allocated to finding bugs [5]–[8], secure software remains decidedly insecure. The NIST National Vulnerability Database tracks an average 14,502 CVEs annually since 2016, and as Figure 1.1 shows, the most severe CVEs constitute a remarkably consistent 15% of all CVEs issued every year. Clearly a different tack is needed. We performed an analysis of the last five years of CVEs issued for the Linux kernel, which is detailed in § 2.2.2, and found that the lack of a *least privilege policy* is the underlying reason that most bugs can be exploited, leading to whole system compromise.

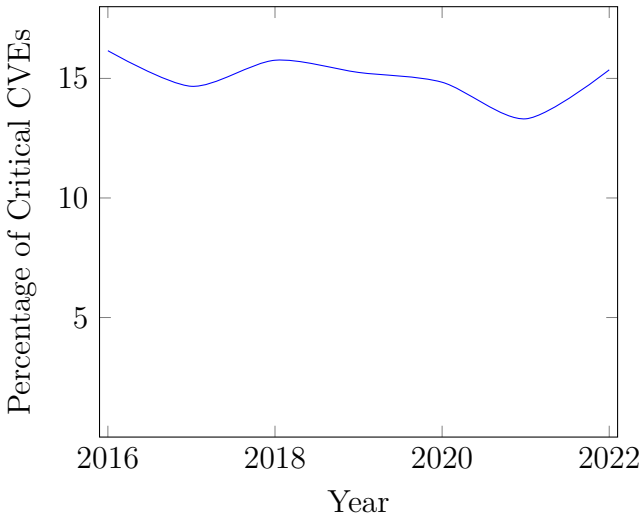


Figure 1.1. Critical CVEs as Percentage of all CVEs have remained consistent for years despite decades of vulnerability mitigation research.

The principle of least privilege states simply that an entity should have access to only the resources needed to complete its task, and no more. Despite it being an idea in computing that dates back to at least the early 1970s [9], [10], some of the most important software projects, such as the Linux kernel, the Apache web server, or OpenSSL, do not enforce the principle of least privilege. Those software systems, like most others, are designed as *monoliths*, which allow any instruction to access virtually any memory address (barring address space-wide restrictions on accessible pages). Any pointer is considered valid and relevant to the current task by the instructions performing the access, regardless of what data the pointer addresses. Least privilege seeks to reduce excess privilege by restricting access only to data (and code) that instructions *must* access to *function properly*. Excess privilege is a security risk, because unintended actions can be performed that destabilize the system, elevated privileges can be assigned to unauthorized users, and sensitive data can be made public.

A classic example of the damage caused by the lack of a least privilege policy is the Heartbleed [11] bug in OpenSSL disclosed in 2014. Heartbleed was a vulnerability in the Heartbeat Extension for TLS, which allows for an established connection to keep alive and test its continued viability. This extension avoids the expensive reestablishment of a secure connection. However, the OpenSSL implementation trusted unverified user-supplied data, and allowed the continuous exfiltration of 64 KB of memory from the target. The contents of that memory often contained private keys, cookie session keys, and user passwords. 24–55% of the Alexa Top 1 Million websites [11] were affected, and, consequently, Heartbleed is considered one of the most catastrophic bugs ever found. However, the Heartbeat Extension does not need access to user passwords or private keys, and in a system designed with least privilege such access would be forbidden. Unfortunately, Heartbleed is not an isolated incident relegated to the past; in just the first three months of 2022, NIST reports over 2,000 high severity CVEs issued.

To address the least privilege problem, I propose Hardware-Assisted Kernel Compartmentalization and Flexible Compartments. Hardware-Assisted Kernel Compartmentalization is a general method of enforcing a compartmentalization policy using new hardware extensions. Flexible Compartments allows for the generation of a compartmentalization pol-

icy that allows for user customization in terms of performance and security which scales to large computer systems. Flexible Compartments utilizes type and call information to generate its compartments, but other avenues exist for compartment discovery. One example, μScope [12], builds compartments based on measurements of memory accesses and control-flow transfers performed during concrete executions. However, dynamic approaches like μScope are inherently imprecise (as they only observe the executed paths through the program), so other approaches should be explored. To that end, I propose IOVec Function Identifier, a semantic similarity framework that operates on stripped binaries, yet is resilient to significant changes to compiler, optimization, or even architecture. Using code similarity, one could develop a compartmentalization policy that is explainable, a trait that is becoming more desirable [13], and more accurate than using only dynamic code interaction measurements.

1.1.1 Threat Model

Throughout this dissertation, the threat model I assume is a strong attacker with full knowledge of the compartmentalization policy in place on the system, and I do not assume probabilistic defenses like KASLR [5] are enabled. I further assume that the attacker has direct access to the target system, and can execute arbitrary programs as their user, and can send any external data, e.g., network traffic or USB data, to the system via arbitrary hardware. However, the attacker cannot change kernel code or install and run their own kernel module. The attacker also cannot modify system-wide settings or active CPU features.

The following sections are brief introductions to each of the three works, followed by my thesis statement.

1.1.2 Hardware-Assisted Kernel Compartmentalization

Hardware-Assisted Kernel Compartmentalization (HAKC) is an enforcement mechanism for arbitrary least privilege policies in the Linux kernel. As the name suggests, HAKC relies on hardware features, and does not expand the trusted computing base via virtualization mechanisms. HAKC uses a novel combination of two ARMv8.5-a (and soon-to-be released

ARMv9) extensions, Memory Tagging Extension (MTE), and Pointer Authentication Code (PAC), to ensure that the current program state allows pointer dereference or control flow transfer. The programmer defines the compartmentalization policy, that catalogs the code and data that belong together in a compartment, and to which compartments a given compartment can transfer control. During compilation, HAKC reads the desired compartmentalization policy, and inserts validity checks before pointer dereference or indirect control flow transfer. If the current program state does not allow the dereference of a pointer or a control flow transfer to occur, HAKC prevents the pointer from being accessed.

MTE allows for the association of a tag (i.e., a color) to a virtual address range. Unfortunately, MTE, like many memory tagging architectures, only allows for a small number of tags. In MTE's case, only 4 bits are used for tags, and 16 compartments are inadequate for large systems like the Linux kernel. Therefore, HAKC introduces a novel two-tiered compartmentalization scheme, where code and data are bundled together into a structure called a Clique, and a set of Cliques are further bundled into a Compartment. Cliques are assigned a color unique to the Compartment (but not unique to the whole system), and each Clique has an access control policy that details what colors the Clique can access. Data is owned by exactly one Clique within a Compartment, and the Compartment and Clique ownership mapping is established and checked using PAC. PAC computes a hash of a pointer value, and a user-specified context parameter, and stores the result in the upper unused bits of a 64-bit pointer. Pointers are signed when memory is allocated using the color and Compartment identifier, and checked at run-time by code in a Clique that it conforms to its data access graph. If control flow leaves the Compartment to another Compartment, data ownership passes to the target through a recolor and resigning operation, which is then undone when control flow returns.

We compartmentalized the `ipv6.ko` and `nf_tables.ko` Linux kernel modules as a proof of concept. We evaluated the performance overhead induced by HAKC using `apachebench` serving several file sizes, and find that the overhead is 1.6%–24% when only the `ipv6.ko` module was compartmentalized. When both modules are used, we measure a reasonable linear growth in overhead. We also simulated browsing Alexa Top Websites that provide an IPv6 address using the compartmentalized `ipv6.ko` module, and found no significant

difference from the unhardened kernel module to retrieve and render webpages and streaming videos.

1.1.3 Flexible Compartments

Creating an enforcement mechanism for compartments is not sufficient for truly effective compartmentalization; the corresponding tight compartmentalization policy must also be created. By compartmentalization policy, we mean the specification of what code and data should belong in the same compartment, what outside data should a compartment be able to access, and what are legal control-flow transitions to outside compartments.

In order to specify a compartmentalization policy, one must know the allowable memory accesses and control-flow transitions. Manual specification of a policy is too cumbersome, because the amount of legal operations in a complex software system are too numerous. Therefore, an automatic solution to compartmentalization policy generation is needed. However, there are two types of data sources one can use to generate policies, static and dynamic data. Static analysis is able to fully identify all possible interactions between instructions and data, such as the set of data a particular instruction can access, or the set of target instructions of a particular indirect jump. However, that analysis is imprecise and over-approximative (due to aliasing), and will therefore add interactions that, in reality, may never happen. Dynamic analysis, however, records only the memory accesses and control-flow transfers that occur during actual observed executions. This analysis provides the lower bound of interactions that are possible within a system, but will not include legitimate code paths and accesses, because they simply were not observed. For example, error code paths are unlikely to be triggered during any particular execution, but need to be accounted for in a compartmentalization policy. Dynamic analysis does provide memory access and control-flow frequency information, and that information can inform compartmentalization policy generation. Placing code and data that frequently interact in the same compartment improves overall performance, because cross-compartment interactions are expensive.

Flexible Compartments is a system for generating performant compartmentalization policies. Flexible Compartments works by generating a weighted directed graph, with edges

between source files in the Linux kernel. The edge weights between two source files contain information that summarizes the level of interaction between them. Static and dynamic information are stored with each edge, and a compartmentalization policy generation can specify how much weight to give to each. By varying the static and dynamic weights, as well as the number of compartments in the final compartmentalization, Flexible Compartments allows the user to specify their security or performance needs, and will automatically generate a compartmentalization policy that meets those needs. We evaluate the performance overhead of several compartmentalization configurations, including varying the number of compartments and static/dynamic edge weight contributions.

1.1.4 IOVec Function Identifier

Flexible Compartments utilizes type and call information to generate compartmentalization policies. Another avenue for compartmentalization policy generation is through code similarity. However, determining how similar two pieces of code are is a difficult problem [14]. One method of determining code similarity is through binary semantic function analysis, which attempts to identify a function's behavior expressed in the source code through investigation of its final binary output.

A lot of work has been done on semantic function analysis, particularly in semantic identification [15]–[27]. Semantic identifiers are either static or dynamic, depending on how they gather the data they analyze. Static identifiers only analyze binary data without executing any instruction in the binary, while dynamic data will execute instructions in the binary, but can also do static analysis. Both techniques have their strengths and weaknesses. Static analysis is quick, relative to dynamic analysis, and its analysis can be complete and correct, because any code path can be explored. However, the incomputability of pointer aliasing [28] and path explosions limits the precision of static analysis. Meanwhile, dynamic analysis is more precise, at the expense of long analysis time relative to static analysis, because the binary is typically executed many times, usually in a virtual machine for safety. Additionally, dynamic analysis is not complete, because some code paths are not traversed.

However, both types of state-of-the-art semantic identifiers share a commonality: they all measure binary code properties. Namely, current semantic identifiers attempt to measure control-flow graph isomorphisms, order and type of instructions, and memory or register access. The assumption is that code property similarity closely correlates with semantic similarity. While it is certainly true that similar code properties indicate similar semantics, measuring different code properties does not indicate different semantics. The same source compiled using different compilers, optimizations, or compiler versions can all lead to arbitrarily different binaries. In other words, the *compilation environment* used to generate a binary strongly determines the output, and the compilation environment information is not preserved in the final binary. For example, Figure 1.2 shows how the `musl` C library implementation of `strlen` produces different assembly when the compilation environment changes. The full disassembly of the various binary versions of this `strlen` implementation can differ by up to 70%.

The fact that such a simple function can produce such different binaries points out a serious flaw in binary property measuring semantic identifiers. There are many ways to perform the same semantic action, and measuring or inferring all possible semantic expressions is impractical. Therefore, when a semantic identifier sees a binary produced by a compilation environment outside of its model, its accuracy is reduced significantly. Since analysts cannot know the compilation environment used to produce a binary, the utility of semantic identifiers that rely on binary properties is limited.

IOVec Function Identifier (IOVFI) is an accurate, compilation environment and architecture agnostic dynamic semantic function identifier. Unlike other state-of-the-art semantic identifiers, which relies on code measurements to determine semantic similarity, IOVFI uses program state transformation sets as its fingerprinting metric. IOVFI automatically finds valid program states for a particular function, and then measures the program state post-execution. The combination of initial program state, and the resulting post-execution program state are called an Input/Output Vector, or IOVec. Since compilers guarantee correct semantic behavior given a source file regardless of compilation environment, the sets of IOVecs generated by IOVFI are highly resistant to changes in compilation environment or even purposeful obfuscation. Using IOVecs, IOVFI can also be used to identify unknown

```
size_t strlen(const char *s)
{
  const char *a = s;
  const size_t *w;
  for (; (uintptr_t)s % ALIGN; s++)
   if (!*s) return s-a;
  for (w = (const void *)s; !HASZERO
   (*w); w++);
  for (s = (const void *)w; *s; s++)
   ;
  return s-a;
}
```

Listing 1.1 Source

```
strlen:
  pushq %rbp
  movq  %rsp, %rbp
  movq  %rdi, -16(%rbp)
  movq  -16(%rbp), %rdi
  movq  %rdi, -24(%rbp)
.LBB0_1:
  movq  -16(%rbp), %rax
  andq  $7, %rax
  cmpq  $0, %rax
  je  .LBB0_6
```

Listing 1.2 `clang-4.0 -O0`

```
strlen:
  movq  %rdi, %rax
  testb $7, %dil
  je  .LBB0_4
  movq  %rdi, %rax
  .p2align  4, 0x90
.LBB0_2:
  cmpb  $0, (%rax)
  je  .LBB0_8
  addq  $1, %rax
  testb $7, %al
```

Listing 1.3 `clang-4.0 -O3`

```
strlen:
  testb $7, %dil
  movq  %rdi, %r8
  je  .L2
  cmpb  $0, (%rdi)
  jne .L4
  jmp .L22
  .p2align 4,,10
  .p2align 3
.L6:
  cmpb  $0, (%rdi)
```

Listing 1.4 `gcc-7.3.0 -O3`

```
strlen:
  testb $7, %dil
  movq  %rdi, %rax
  je  .LBB0_4
  movq  %rdi, %rax
  .p2align  4, 0x90
.LBB0_2:
  cmpb  $0, (%rax)
  je  .LBB0_8
  incq  %rax
  testb $7, %al
```

Listing 1.5 `clang-3.9 -O3`

Figure 1.2. Binary differences caused by changing compilation environments

binary versions based on function behavior, and detect major semantic differences between different versions of known binaries. As a first in class feature, IOVecs generated using one architecture can be used to identify functions in another architecture through the use of a small translation layer.

IOVFI utilizes a guided, mutational fuzzer to generate IOVecs, although other schemes could be used. Fuzzing, which rapidly and repeatedly executes code with a randomized program state, is ideal for analyzing unknown binaries, because no *a priori* knowledge is required. For every function in the binary, IOVFI randomizes the input arguments, and begins executing the instructions. When the function returns, IOVFI records the program state, and saves the initial input and the resulting program states, along with the unique system calls made during execution, into an IOVec. This process continues until a user-specified amount of code coverage is achieved, or the fuzzer has run for a user-specified amount of time. IOVFI also automatically determines which input arguments are intended to be pointers, and preserves that information in the IOVec. When a pointer is detected, during the initial fuzzing step, the target memory area is also fuzzed. When IOVFI completes its training phase, IOVecs are sorted into a binary tree, with known functions as leaves. This sorting later allows for quick classification of unknown functions.

We evaluated IOVFI on F-Score (harmonic mean of precision and recall) accuracy, and found that it achieves an overall average of .779 $\pm$.0777. When the compilation environments of the IOVec binaries differ from those of the unknown binaries—the situation in which current state-of-the-art semantic identifiers find most difficult—IOVFI achieves a .766 $\pm$.0682 average F-Score. IOVFI's F-Score accuracy shows a 94% to 39% improvement over the static *BinDiff 6* analyzer, in overall and differing compilation environment scenarios respectively. We significantly outperform the state-of-the-art `asm2vec` static analyzer in all but the matching compilation environment case. Evaluating against dynamic semantic identifiers, IOVFI is 25%–53% more accurate than the state-of-the-art. When identifying functions in purposefully obfuscated binaries, we are 39.3% more accurate than dynamic identifiers. Finally, IOVFI achieves a similarly high accuracy of .811 when identifying `AArch64` functions using `X64` IOVecs.

1.2 Thesis Statement and Dissertation Layout

With the problems of least privilege and semantic identification described, and my work outlined, I arrive at my thesis statement:

External (potentially abstracted) program state aids in the discovery, establishment, and enforcement of compartments in monolithic computer systems.

By external program state, I mean program state that is established or maintained outside of the control of the executing program. In particular, external program state includes the memory tagging functionality provided by MTE, and the IOVecs generated by IOVFI. The tags maintained by MTE are required to provide Clique ownership information at runtime, yet are only accessible through specific instructions and not through pointer dereference. By design, IOVecs establish an external program state outside the influence of the target function to detect semantic similarity, as well as quantify the similarity of disparate functions.

This dissertation provides three works that utilize program state in novel ways to support the compartmentalization of computer systems. The following chapters will detail HAKC, FlexC, and IOVFI respectively, as well provide experimental results, discussion, and related works for each. Taken together, the works provide strong evidence in support of the thesis statement.

2. HARDWARE-ASSISTED KERNEL COMPARTMENTALIZATION

2.1 Introduction

Modern kernels have expanded their functionality well beyond the "three easy pieces" of concurrency, virtualization, and persistence [29]. Users expect additional features, such as protocol implementations, advanced filesystems, and driver support for an ever growing number of devices. However, as the kernel has grown, its monolithic design, which provides only a single address space for all kernel functionalities, persists. With the increased size and complexity of the kernel, the number of CVEs issued for Linux per year has grown by over 270% from 2005 to 2020.

Loadable kernel modules (LKMs), which are the mechanisms through which additional functionality gets added, provide a natural compartmentalization boundary for the kernel. As kernel module code can always be compiled into the main kernel image, at the source level, there is no clear difference between core kernel code and LKM code. However, when compiled as individualized units, LKMs are not part of the core kernel image that gets loaded by the bootloader. Instead, the kernel dynamically loads an LKM when the kernel needs the particular functionality offered by the LKM. Thus, at runtime, there is a clear and logical separation between LKMs and the rest of the kernel, but the monolithic design of the kernel effectively erases that separation [30].

Bugs in LKMs become just as severe as other kernel bugs, as all code and data exist in the same address space, with no isolation and executing with elevated privilege. Unfortunately, the sheer size and complexity of LKM code create an attack surface much larger than the core kernel. Of the 567 high severity CVEs we analyzed (see § 2.2.2), 301 were found in the `drivers/` and `sound/` directories (or contained the word "driver" in the CVE description for cases of proprietary code, such as the Nvidia GPU driver). We argue that most of the code in those directories are intended for LKMs, and thus that most high severity CVEs come from LKMs, despite our underapproximation of CVE sources.

As an example of a high severity Linux CVE, consider CVE-2016-4997 [31] listed in Listing 2.1. The code is part of the IPv4 packet filtering subsystem, and is executed during

```
1 static void compat_release_entry(struct compat_ipt_entry *e) {
    struct xt_entry_target *t;
3   struct xt_entry_match *ematch;

5   /* Cleanup all matches */
    xt_ematch_foreach(ematch, e)
7     module_put(ematch->u.kernel.match->me);
    t = compat_ipt_get_target(e);
9   module_put(t->u.kernel.target->me);
  }
```

Listing 2.1 Packet filter code that allows root access. A user-controlled pointer value can be passed to `module_put` in line 7 without violating memory safety or control-flow integrity, and an underlying integer is decremented.

error cleanup. The exploit involves the attacker supplying a small positive integer value via a system call, which, due to the LKM only performing an upper bound test and not a lower bound test, can lead to a corruption of a structure submember used as an offset value in a pointer computation. The pointer, computed using the offset submember corrupted by the user, is then written to the `me` pointer in line 7, which decrements an underlying integer, allowing an arbitrary kernel integer (e.g., the current process `UID`) to be decremented.

This exploit is an example of a data-only attack that allows for accessing data beyond what the developer intended, which the monolithic design of the Linux kernel happily allows. A properly executed exploitation of this CVE does not violate memory safety, as all memory accesses are in validly allocated and live memory regions, and no practical memory safety mechanisms [32] prevent submember corruption, of which this exploit takes advantage. Control-flow integrity [33] is not violated either, because execution flows along a valid path at all times. Consequently, while existing mitigations such as memory safety and control-flow integrity have a place in securing the kernel, compartmentalization is necessary for truly secure kernels. Compartmentalizing the packet filtering functionality so that accessible memory is restricted to only that which the developer intends prevents such data-only exploits, even in the presence of buggy code. In that way, compartmentalization provides similar security guarantees to microkernels, but without the significant engineering changes that microkernels impose.

Current state-of-the-art commodity kernel protections (as opposed to embedded kernel defenses [34], [35]) generally fall into one of three categories: virtualization-based, microkernel-based, or compiler-based. Virtualization-based protections [36]–[39] employ a hypervisor to monitor execution or provide stronger isolation between execution domains. Microkernel-based protections [40], [41] completely redesign the operating system to minimize the Trusted Computing Base (TCB) to typically include only the virtual memory management and IPC, and isolate other traditional kernel services as user-space processes. Compiler-based protections [5], [32], [33] introduce security checks or randomization [42] by the compiler that attempts to thwart code-reuse or data-only attacks. Virtualization and microkernel defenses provide the strongest protections, but are the least performant, and still rely on additional software TCB. Compiler protections, with the exception of KASAN [32], are more performant, but only protect a subset of the attack surface, as with kCFI [33], or are often circumvented [43].

In this paper, we present Hardware-Assisted Kernel Compartmentalization (HAKC), a mechanism for compartmentalizing kernel code and data. HAKC relies on hardware features for enforcement, which avoids growing the TCB, yet provides strong data and control-flow protection. HAKC splits code and data into partitions that contain a developer-specified mix of both, and collects the partitions into a larger grouping for efficient policy enforcement. A data-access policy is specified for each partition within the larger group, and a control policy is defined for the larger group if control flow needs to exit its constituent partition set. HAKC provides fine-grained data-access and control-transition policies to prevent arbitrary data access and code execution. The data-access policy ensures that all data belongs to exactly one partition, and the accessed data conforms to the data-access policy defined for each partition. The control-transition policy checks that indirect control flow targets also conform to the partition set access policy. When control flow exits the partition set, data ownership is transferred to the target, and then restored upon return. In this way, HAKC optimizes and enforces safe local data access; code and data access within a partition is secure and quick, relative to those outside. However, data and code defined outside the partition is accessible, but only if explicitly needed. While we designed HAKC around

compartmentalizing LKMs, it is not limited to only that use case; HAKC can be applied to core kernel code, as well as user-space code.

We implement Hardware-Assisted Kernel Compartmentalization for the `ipv6.ko` and `nf_tables.ko` LKMs in Linux 5.10.24 using hardware features present in the ARMv8.5-A ISA. We measure the performance overhead of compartmentalizing `ipv6.ko` using microbenchmarks, and find that HAKC imposes an average 1.6%–24% overhead. Additionally, we measure the overhead of using two compartmentalized LKMs together, and find that the overhead grows linearly. Finally, when simulating typical browsing behavior using the Alexa Top websites, we find no significant difference using our compartmentalized LKM over an unmodified LKM. To summarize, this paper provides the following contributions:

- A compartmentalization policy API for defining fine-grained compartmentalization policies.
- A practical, hardware-based compartmentalization enforcement mechanism.
- An implementation of a compartmentalization policy on the `ipv6.ko` and `nf_tables.ko` LKMs[1].
- An extensive evaluation on the overhead imposed by our compartmentalization policy and enforcement, demonstrating its practicality.

2.2 Background and Motivation

Here, we present some background about Pointer Authentication (PAC) and Memory Tagging Extension (MTE), the hardware security primitives we used to build our prototype HAKC implementation. Both PAC and MTE are present in the ARMv8.5-A ISA. We additionally provide an analysis of high severity CVEs that motivate the need for HAKC.

[1]↑Available at https://github.com/mit-ll/HAKC

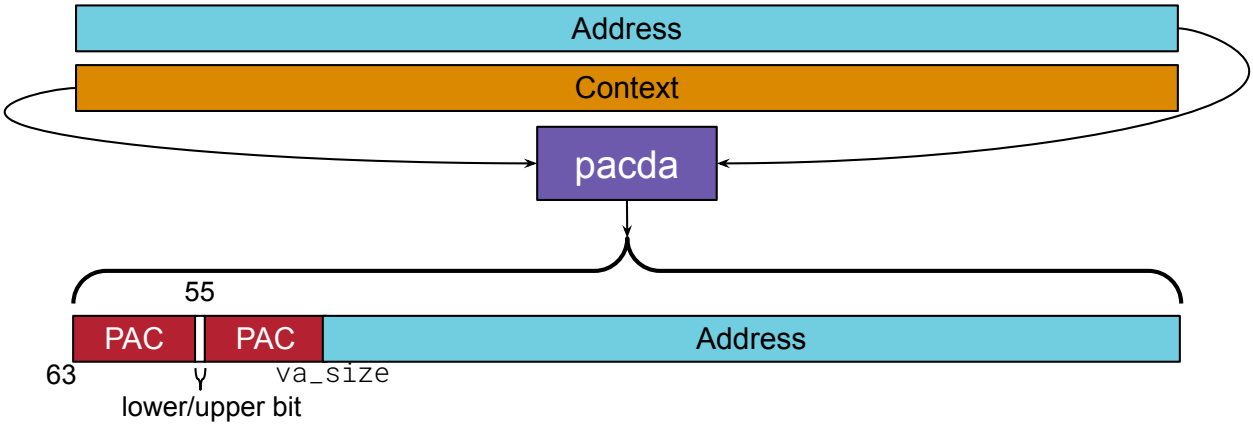


Figure 2.1. Pointer signing using PAC. The upper/lower bit indicates if higher bits are used in the PAC signature.

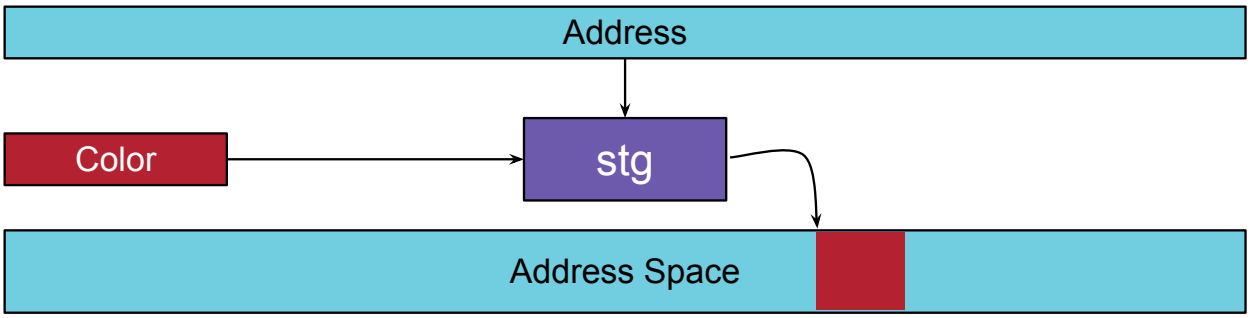


Figure 2.2. Address space coloring using MTE.

2.2.1 Hardware Primitives

Pointer Authentication

Introduced in ARMv8.3, Pointer Authentication is used to cryptographically sign pointers, and store the signature in the "unused" upper bits of a 64-bit pointer (see Figure 2.1).

PAC implements two instruction classes, one for signing and one for authenticating a signed pointer, and allows for using five different keys, two for data and code pointers each and one user-specified key. For example, in Figure 2.1, the `pacda` instruction specifies using the `a` key for signing `d`ata pointers. Signing involves specifying a pointer to be signed, the key to use for signing, and a 64-bit *signing context*. PAC was initially designed to mitigate code reuse and pointer substitution attacks [44], because the signed pointer no longer references validly mapped memory, and an invalidly modified pointer will fail future authentication. The attacker, therefore, will have to guess a valid signature for a replacement pointer, which is hard because the signature uses the cryptographically secure QARMA block cipher [45]. However, separate address space domains can be established by varying the signing context, because it can be any 64-bit value. For instance, the stack pointer can be used as the context to ensure stack-based buffer overflows do not overwrite valid return addresses with attacker controlled values. Liljestrand et al., implemented a type safety mechanism by using an object ID as the PAC context [46], and Farkhani et al., implemented a temporal memory safety mechanism using allocated object metadata as the context [47]. Other uses for PAC have been proposed [48], [49], and HAKC uses PAC (combined with MTE) to enforce compartments' access policies.

To obtain a valid pointer, the signed pointer must be authenticated using the same key and context. If either the pointer (*sans* signature) or the context are different from the values used during signing, the authentication results in yet another invalid pointer. If the pointer, key, and context are the same values used during signing, the signature is stripped from the pointer, and the original (presumably valid) pointer value is restored. HAKC relies on this behavior to compute a context that was expected to sign a particular pointer, using a combination of information known at compile time and gathered during runtime.

Memory Tagging Extension

Memory tagging extension (MTE), introduced in ARMv8.5-A [50], allows for assigning a "color" or tag to a memory region, which can be used to segregate the address space into distinct regions. MTE introduces two instruction classes, one to assign a color to a memory region (shown in Figure 2.2), and one to retrieve the current color of a memory address. Given infinite tags, one could very simply create highly compartmentalized code — each compartment could be individually colored. However, MTE imposes some constraints on how it can be used: only 16 colors are available for use, the memory address must be aligned to 16 bytes, and the region to be colored can be no smaller than 16 bytes. The limited number of colors available makes simplistic compartmentalization inadequate, because the compartments are too broad in scope. The attack surface of only 16 compartments in the Linux kernel is large enough that bugs are unlikely to be mitigated. However, as this paper will show, the combination of PAC with MTE allows for the creation of significantly more compartments than the available colors. Colors are reused, but compartments are protected using PAC contexts computed from hard-coded values known at compile time and from the address tags retrieved during runtime in order to prevent reused colors from enabling spurious access.

2.2.2 Kernel Vulnerability Analysis

As inspiration for HAKC, we analyzed high severity (CVSS 3.0 rating 7.0 or higher) CVEs issued for the Linux kernel from Jan. 2015 through May 2021, and determined if the CVE could be mitigated with memory safety or compartmentalization. A CVE is considered mitigable if the presence of memory safety and/or compartmentalization would prevent the bug, and unmitigable if neither mechanism would prevent the bug. The basis for classification was determined by searching for keywords in the description that map to the two defense mechanisms (i.e., *arbitrary code execution* maps to compartmentalization, while *use-after-free* maps to memory safety), or manual analysis of patches in cases where the description was unclear. Figure 2.3 presents a summary of our findings. Out of the 567 CVEs in our dataset, 229 could be mitigated through compartmentalization, and 193 could be mitigated

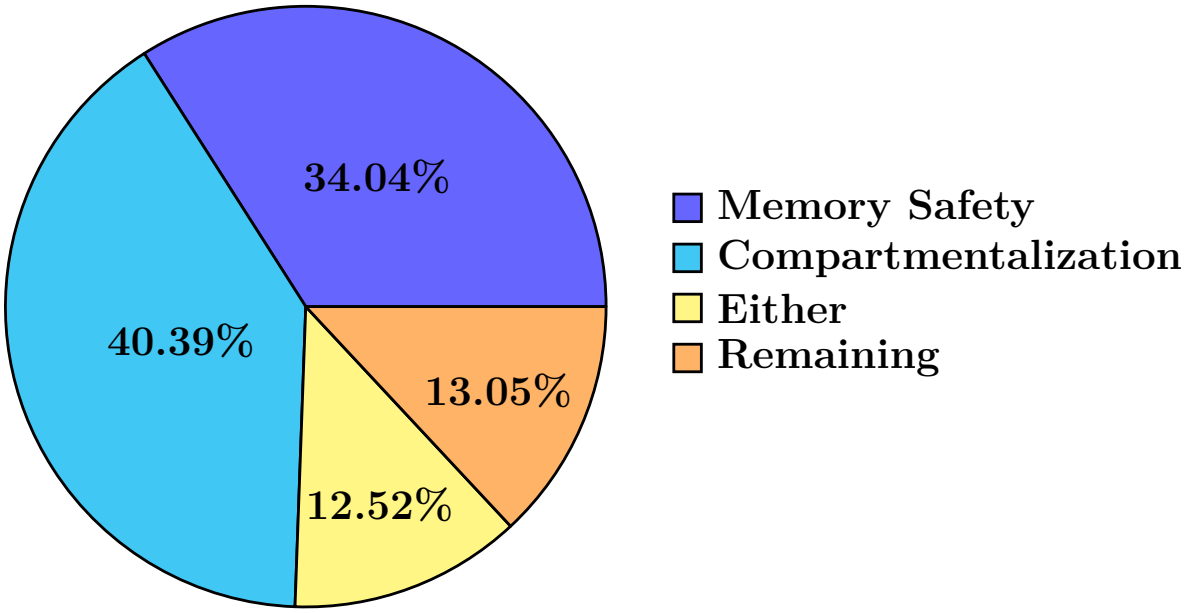


Figure 2.3. A breakdown of mitigations for Linux kernel high severity CVEs. "Either" indicates that if memory safety *or* compartmentalization were present, the bug would not be exploitable. "Remaining" indicates that memory safety or compartmentalization would not mitigate the CVE.

using memory safety. Only 71 could be mitigated by either defense mechanism, implying the continued importance of memory safety alongside compartmentalization, and only minimal overlap in protection when both are enabled. There are 73 CVEs that are unmitigable with compartmentalization and memory safety, of which 57 involve incorrect or missing domain-specific logic, such as discarding returned error values or a cold path missing a data validity check. The remaining unhandled CVEs involve race conditions (9), integer over/under-flows (8), and a configuration that enables unsupported functionality (1).

2.3 Threat Model and Assumptions

In line with other kernel security mechanisms, we assume that an attacker does not have root access, and thus cannot modify kernel modules. However, they can take arbitrary actions in attempt to compromise a victim kernel module, including making arbitrary system calls or having peripherals send arbitrary data [51]. We also assume that the LKM itself is not malicious, but contains exploitable bugs. Kernel functionality outside of the victim

```
static unsigned long *m1_counts;
typedef struct msg {
    long idx;
    unsigned long val;
} msg_t;

unsigned long m1_get(msg_t* m) {
    return m1_counts[m->idx]; // idx not checked -> ARBITRARY READ
}
EXPORT_SYMBOL(m1_get);

int m1_init(void) {
    m1_counts = kmalloc(SIZE*sizeof(unsigned long));
}
```

Listing 2.2 LKM 1 (Arbitrary Read)

```
static unsigned long counts[SIZE];
extern unsigned long m1_get(msg_t*);

static int m2_ioctl(struct inode *inode,
                    struct file *file,
                    unsigned int ioctl_num,
                    unsigned long ioctl_param) {
    msg_t *tmp;

    switch(ioctl_num) {
        case MSG_PUT:
            tmp = (msg_t*)ioctl_param;
            counts[tmp->idx] = tmp->val; // idx not checked -> ARBITRARY WRITE
            break;
        case MSG_GET:
            tmp = (msg_t*)ioctl_param;
            tmp->val = m1_get(tmp);
            break;
        default:
            return FAILURE;
    }
    return SUCCESS;
}
```

Listing 2.3 LKM 2 (Arbitrary Write)

LKM is part of the trusted source base, and we assume that data originating from the kernel is valid. Trusting data passed into the LKM could lead to a confused deputy attack, but preventing such an attack would require full code and data flow analysis in the kernel. Such an analysis is currently impractical, and thus we require the kernel to be a trusted agent (similar to the trusted core of a microkernel). Additionally, we include the core SoC in the trusted computing base, including its tagging and pointer authentication implementations, but IO devices are outside of our trusted components and can be malicious. Three exceptions to our hardware assumption, however, are direct memory access (DMA) actions, hardware

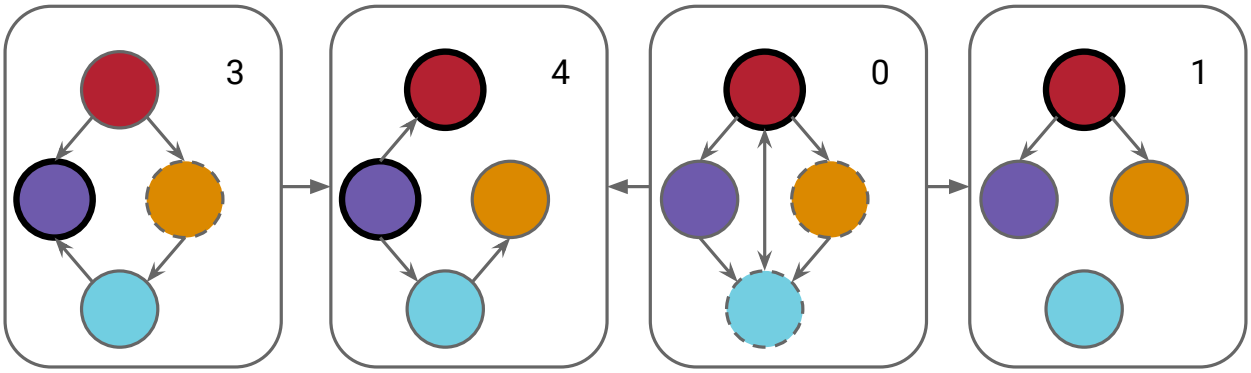


Figure 2.4. An example compartmentalization involving four Compartments, each of which contains four Cliques. Edges between Compartments are allowable transitions. Bold Cliques are valid Compartment entry points, and dotted Cliques are valid Compartment exit points.

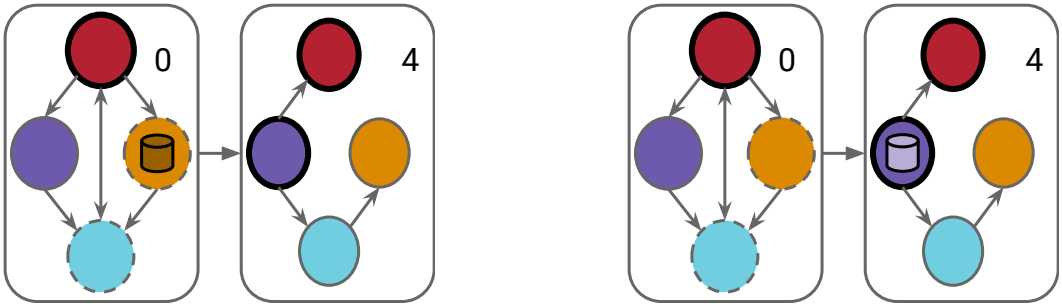
glitching attacks [52], and side channel attacks, such as Spectre [53], Meltdown [54], or Rowhammer [55].

We do not assume any further virtualization or security layer that provides a level of trust, such as a hypervisor or verified microkernel. Instead, HAKC moves policy enforcement to the hardware, and removes the difficult problem of verifying trusted software [56]. HAKC is designed to run on bare metal, but is capable of running in a virtual machine provided an existing implementation of hardware features.

Listing 2.2 and Listing 2.3 is an example of two partial LKM implementations that conform to our threat model, but provide an arbitrary read and write. Listing 2.3 is dependent on Listing 2.2, and the programmer's intent is to only read from the defined arrays. However, due to a missing check on `idx`, if the user calls `ioctl` with `MSG_PUT` or `MSG_GET`, and an index outside the range of $[0, \texttt{SIZE}]$, then any address can be written or read (barring page permissions). The monolithic design of the kernel will simply allow these accesses, but HAKC prevents them by compartmentalizing the two LKMs.

2.4 HAKC Compartmentalization API and Enforcement

HAKC is built around two core contributions, which, when combined, are instrumental to its ability to establish isolation within the kernel without further virtualization: the



(a) Pre-Compartment Transition (b) Post-Compartment Transition

Figure 2.5. An example Compartment transferring data ownership to an external Compartment. During this Compartment transition, orange data is recolored purple, and the data ownership is moved to the target Clique in the external Compartment. Upon return, the data is colored orange, and data ownership is restored.

Compartmentalization Policy API and a hardware-based *Compartment Enforcement Mechanism*. The Compartmentalization Policy API exposes primitives that allow the developer to establish a fine-grained compartmentalization policy on code and data, while the Compartment Enforcement Mechanism efficiently enforces the specific compartmentalization policy at runtime. We detail each here.

2.4.1 Compartmentalization Policy API

The HAKC Compartmentalization Policy API allows developers to assign code and global variables to compartments. Stack and heap variables are assigned to the same compartment as the code that allocates them. The compartmentalization policy also specifies allowed control flow between compartments. When control flow transitions between compartments, any required data is also automatically transferred.

The HAKC compartmentalization policy API allows users to configure the following in HAKC's novel two-level compartmentalization scheme:

1. *Cliques*, a partitioning of code and data into one or more groups, with each function and data object belonging to exactly one partition.

2. *Compartment*, a second level grouping of at least one Clique, with each Clique belonging to exactly one Compartment.

3. *Clique Access Policy*, the set of Cliques within a Compartment that a particular Clique can access, including itself.

4. *Compartment Transition Policy*, the set of Cliques that can legally transition control to outside the Compartment, and the set of allowable external Cliques that are valid control-flow targets.

Figure 2.5 shows two example Compartments. The red Clique in Compartment 0 can access the red, purple, blue, and orange Cliques, while in Compartment 4, red can only access itself. Figure 2.4 illustrates an example of several Compartments, and the allowable transitions; Compartment 0 can transfer control flow to Compartments 1 and 4, but Compartment 3 can only transfer to Compartment 4.

The two-level compartmentalization policy API introduced by HAKC provides three key benefits: 1) the ability to create a large number of compartments with a limited number of colors — overcoming the classic limitation of few tag bits; 2) efficient access to data defined in a Compartment, i.e., local data optimization; and 3) the flexibility for the developer to make fine-grained security/performance trade-offs. The first benefit frees the developer from practical limitations present in any commodity tagged hardware when designing compartmentalization policies. The number of desired compartments will exceed the number that any hardware-only system can supply (e.g., 2^{tag_bits}), inspiring our new two-level scheme that allows tag bit reuse across Compartments. As we will show later, grouping Cliques into a Compartment allows for creating several orders of magnitude more compartments than available tags would otherwise allow.

The second benefit allows for easier policy creation, and more efficient policy validation checks. While a Clique is executing, any data that is accessed by a pointer must satisfy two conditions: 1) the data must belong to the Compartment in which the Clique resides; and 2) the data must belong to a Clique the current Clique is allowed to access under the Clique access policy. These conditions are checked at runtime prior to the first dereference of a

pointer in a function, but are not checked again unless the pointer is modified. Satisfying these two conditions ensures that arbitrary data access is prevented, and that data ownership is enforced. These conditions also enable faster checks as only the Clique access policy needs to be checked, and that policy only concerns 2^{tag_bits} Cliques, allowing a highly optimized implementation compared to the Compartment access policy.

To illustrate the third benefit — fine-grained security/performance trade-offs — we first describe how Clique and Compartment access policies work. The developer establishes a Compartment by partitioning code and data into one or more Cliques, determining which Cliques each particular Clique should legally access, and which Compartments are valid control-flow targets. The specific compartmentalization policy can be determined manually, or automatically through static or dynamic analysis, and can be different for different Compartments. All data and code in a Compartment must belong to exactly one Clique, and directed edges between Cliques indicates valid access. The directed edges represent a forward-edge Clique-based control-transfer policy, and does not need to be symmetric. For instance, the developer might want a green Clique to call a function in a red Clique, but forbid the red Clique from calling a green Clique.

When control flow must exit a Compartment, through either a direct or indirect function call, then the ownership of data that exits must be transferred to the target destination, and then restored upon return. The transfer ensures that data checks in Cliques can proceed as intended, which maintains valid data ownership. In the case of indirect function calls, the target is checked to ensure that it conforms to the valid transition policy that the Compartment defines, and that the target is a valid entry to the target Compartment. If the target of an indirect call is within the same Compartment, the access policy for the current Clique must be followed, but no data ownership is transferred. The control-flow checks ensure a valid control path is followed, and arbitrary code execution is prevented.

By adjusting how code and data are partitioned into Cliques and Compartments, the developer can make fine-grained trade-offs between security and performance. As the number of Compartments increases, i.e., the more compartmentalized the kernel becomes, the harder an attack becomes, because the attacker has to find a valid control-flow path that obeys both the Clique access policy and Compartment control-transfer policy. However, the increased

Compartment count necessarily leads to more data ownership transfers, which can incur large performance overhead. HAKC allows developers to specify fine-grained boundaries to suit their particular performance and security needs.

2.4.2 Compartmentalization Enforcement Mechanism

While Clique code is executing, HAKC does not rely on any additional TCB, but instead uses hardware for access policy enforcement. Prior compartmentalization mechanisms rely on additional layers of abstraction, e.g., kernel code for user-space compartments [57]–[59], or hypervisors for kernel code [36], [60]–[62]. Breaking this "turtles all the way down" paradigm for compartmentalization by rooting trust in hardware avoids adding layers of abstraction and growing the TCB. HAKC is the first to solve this challenge for realistic, commodity hardware.

In order to provide compartmentalization, HAKC needs to be able to partition the virtual address space separately from traditional paging, and the ability to associate a pointer with metadata of bit-size larger than the available address space partitions, referred to as the pointer's *conjoined metadata* (CM). By virtual address space partitioning, we mean a method of designating a virtual memory address range as distinct from the rest of the address space. Tagged architectures, which typically provide a small number of bits to associate with (or *color*) a virtual memory address range, are an existing partitioning method. All pointers, either statically created at load time or allocated dynamically, are associated with a specific CM encoding which Clique owns the underlying data. The association must be hard to compute given the pointer and CM. HAKC enforces partition access policies by, for every pointer accessed during runtime, computing a *candidate CM* using the address space partition information (which equates to Clique membership), and Compartment information. Before the pointer is accessed, the candidate CM is compared with the pointer's actual CM. If the runtime information causes the candidate CM to differ from the actual CM, then the pointer dereference cannot happen.

As long as the hardware provides address space partitioning and pointer CM association primitives, then HAKC guarantees that the defined compartmentalization policy is followed.

If some bug inside the compartment modifies a pointer such that it points to data that violates the access policies defined for either the Clique or Compartment, the candidate CM will differ from the pointer's CM. Accordingly, if some bug outside the Compartment modifies a pointer that is currently being used by the Clique to violate access, the candidate CM will again differ from the correct CM. In both cases, data access is prevented, and compartmentalized code is prevented from accessing data not explicitly granted to it. While we implemented HAKC using ARMv8.5-a, HAKC is not tied to any specific architecture. Any mechanism that provides the necessary partitioning and association primitives may implement HAKC.

2.4.3 Example Case Study

Here we describe how HAKC can compartmentalize the two LKMs listed in Listing 2.2 and Listing 2.3. In this example, all code and data in Listing 2.2 will be in the same Black Clique in Compartment 1 (referred to as $(1, Black)$), while the code and data in Listing 2.3 will be in the Gold Clique in Compartment 2 ($(2, Gold)$).

When `m2_ioctl` executes, it first validates that `tmp` is accessible by checking its $(2, Gold)$ CM with the candidate CM computed with runtime data. `m2_ioctl` will also check if the $(2, Gold)$ CM for `counts + tmp->idx` matches the computed candidate CM, and, if `m1_get` is called, will recolor `tmp` to black, and associate its value with $(1, Black)$. When `m1_init` executes, it associates `m1_counts` with $(1, Black)$. Finally, when `m1_get` executes, it checks `m` and `m1_counts + m->idx` with $(1, Black)$.

If an attacker directs control flow to `m2_ioctl`, then `ioctl_param` must be properly associated with $(2, Gold)$, which is computationally hard to perform. Similar conditions must be satisfied for `m1_get`. Additionally, if a bug outside of $(1, Black)$ changes the value of `m1_get` to point outside the Compartment, then the pointer will not be associated with $(1, Black)$, and HAKC prevents its dereference.

2.5 Compartment Policy and Enforcement Mechanism Implementation

Here, we detail how HAKC enforces the data and control-flow policies that provide isolation to compartmentalized code. The HAKC Compartment Enforcement Mechanism uses a combination of tagged architecture and cryptographic hashes to provide access enforcement. Namely, the Compartment Enforcement Mechanism uses Arm's MTE to provide tagging support, and PAC to provide cryptographic hashing. See § 2.2 for details. PAC is used to ensure that pointers have not been tampered with inadvertently, and that they conform to the various access-control policies defined for Cliques and Compartments, while MTE provides the runtime Clique membership. By combining information known at compile time, such as a Compartment identifier and access-control policies, with MTE colors gathered dynamically, HAKC can provide significantly more compartmentalization granularity than the 16 compartments natively provided. HAKC recycles colors in different Compartments, but the compile time information effectively creates "hues" of the available colors to allow for a large number of compartments.

Cliques

A Clique belongs to exactly one Compartment, and combines code and global, stack, and dynamically allocated data into a logical group, all of which is assigned a color, C_c. C_c needs to be unique to the Compartment the Clique belongs to, but it does not need to be globally unique. In fact, one of the primary contributions of this paper is a design that allows for the safe multiplex use of colors in different compartments. A good example of the type of information that a Clique contains is what is defined in a typical C source file: exported and static functions and global variables, stack allocated objects, and dynamically allocated memory. We do not, however, force all functions or data in a source file to belong to the same Clique, and the developer is free to partition as they see fit.

Using their specific C_c, Cliques also define two tokens, Tok_{acl} and Tok_s, used in authenticating a pointer and signing pointers respectively. Tok_{acl} is used to generate the context used for PAC authentication, and encodes both the Clique's Compartment identifier, ID_n, and allowable Clique code and data accesses. Tok_s encodes ID_n and C_c, and provides the

Table 2.1. Clique and Compartment properties.

Clique	Compartment
Access Token (Tok_{acl})	Entry Token (Tok_{ent})
Signing Token (Tok_s)	Valid Compartment Targets (T_n^*)
Color (C_c)	Unique Identifier (ID_n)

PAC context when signing pointers. We will detail how these tokens are used to enforce compartments in § 2.5.1.

Compartments

A Compartment consists of at least one Clique but no more Cliques than the number of available tags, N_{tag}, and is assigned a globally unique identifier, ID_n. All data accessed by any Clique must belong to the Compartment, and the identifier is used to ensure that is the case during pointer authentication.

Additionally, a Compartment defines an entry token, Tok_{ent}, which encodes the Cliques that can be targets of indirect jumps, along with ID_n. Tok_{ent} is known to all other Compartments that could potentially execute code in a Clique. Similarly, a Compartment must know all valid potential Compartments to which it could transfer control flow. Therefore, a Compartment maintains a mapping of valid Compartment ID_n and the respective Tok_{ent} in T_n^*. Before an indirect call is executed, the target function is checked that it belongs to a valid Compartment, and is a valid entry Clique using each entry token in T_n^*. Figure 2.4 is an example of several Compartments, along with their allowable Compartment transitions. In this instance, control flow is able to transfer from Compartment 0 to either Compartment 1 or 4, but not Compartment 3. Additionally, if control flow is going from Compartment 0 to Compartment 4, the target Clique *must* be orange or red, and cannot be green or purple. Cliques in different Compartments have different access-control policies, yet share colors.

Data Access Policy

All data accessed by a Clique through a pointer must belong to the currently executing Compartment, and be validly accessible according to the current Clique access-control policy.

PAC is used to validate both conditions hold. Pointers are signed using Tok_s, and when authenticated later, Tok_s is calculated using Tok_{acl}, and the pointer target MTE color. If the pointer is erroneously manipulated, or points to data that is either the incorrect color or does not belong to the Compartment, then the computation of the PAC authentication context will differ from Tok_s and will thus fail authentication.

Compartment Transitions

When a Compartment needs to transfer control to another Compartment, two actions need to occur: 1) the target must be validated against T_n^*; and 2) data ownership must be transferred to the target destination. In the event of a direct function call to outside the Compartment, only step 2 needs to occur.

Function pointers are signed by their Clique using their Tok_s, and, similar to how data pointers are authenticated, that same token is computed using the target MTE color and the Tok_{ent} from T_n^*. Each Tok_{ent} in T_n^* is used to compute a possible PAC authentication context. If the authentication of the signed function pointer succeeds with the computed context, then control flow is allowed to the target Clique in the different Compartment. If authentication fails, then a different Tok_{ent} is tried until all valid transitions are exhausted, at which point control flow to the target is prevented.

If the Compartment transition is allowed, all pointer arguments (and any submember pointers) are recolored the target Clique color, and the pointers are resigned using the target Clique Tok_s. The act of recoloring and resigning pointers transfers ownership of data from the current Clique to the target Clique, and pointer validity happens within the new Compartment as previously described. When the indirect call returns, data ownership is restored to their previous state prior to the indirect call.

We note that only transferred pointers are resigned (and retagged). No effort was made to automatically invalidate any aliases of transferred pointers. This means that lingering aliased pointers in the origin Clique could allow access to data owned by another Clique of the same color, because the original signature would be valid. Pointer struct members and pointers to objects allocated on the current stack frame are resigned and protected, but

Table 2.2. Computed PAC context used for access enforcement for a pointer `p` used by Clique M.

Operation	PAC Context
Transfer to M	$Tok_{s,M}$
Data Access Check	$Tok_{acl,M} \wedge V(C_p)$
Valid Transfer to Compartment G	$Tok_{ent,G} \wedge V(C_p)$

HAKC does not solve the general aliasing problem. We rely on the programmer to invalidate aliased pointers when necessary.

2.5.1 Access Enforcement

For both signing and authentication, PAC takes as input a pointer and a user-specified 64-bit context. For a signed pointer to pass authentication, the *exact* context used to sign the pointer must be provided. In order for HAKC to ensure that the signing context can be correctly provided when authenticating, all tokens (e.g., Tok_{acl} or Tok_{ent}) have the same general form:

$$Tok_{i,n} = ID_n \oplus (V(C_i) \vee \cdots) \tag{2.1}$$

where $V(C_i)$ is a bitvector of size N_{tag} with the bit corresponding to color C_i set to 1 and all other bits set to 0. In other words, HAKC tokens consist of the particular Compartment identifier concatenated with one or more vectorized color values bitwise `OR`'d together. The composition of colors with Compartment identifiers is what allows for the reuse of colors in different Compartments, and the creation of far more compartments than N_{tag}, while still providing strong compartmentalization guarantees. The number of compartments potentially available is $2^{64-N_{tag}} \cdot N_{tag}$, which for MTE ($N_{tag} = 16$) equates to $4 \cdot 10^{15}$ compartments. Dynamically allocated data is immediately transferred to the Clique which created it, and stack-allocated data passed to compartmentalized functions must be transferred to the current Clique if not proven safe (see § 2.5.4). All other data must have been previously signed, either by the kernel when control first flows into the Compartment, or by other Cliques during a cross Compartment transfer.

Signing tokens, which always belong to a specific Clique, utilize only one vectorized color: their own. Access tokens (i.e., Tok_{acl} or Tok_{ent}) are comprised of the relevant allowable colors. For example, in Figure 2.5, the purple Tok_{acl} in Compartment 0 consists only of the vectorized blue and purple values `OR`'d together, while the Compartment 0 red Tok_{acl} has all four color values set to 1. Similarly, in Figure 2.4, Tok_{ent} for Compartment 4 will be composed of the vectorized purple and red, as those are the valid entry Cliques to that Compartment (indicated with bold outlines).

To confirm that a Clique can access data at a signed pointer, a candidate signing token is formed by computing the color vector of the underlying pointer data, and concatenating ID_n. The candidate signing token is then bitwise `AND`'d with Tok_{acl} to provide the PAC authentication context. If the Clique is allowed access to the pointer color, and the pointer was signed by the current Compartment, then the bitwise operation results in the exact context used to sign the pointer, and PAC authentication succeeds. If data belongs to a different Compartment, but is colored an accessible color, PAC authentication fails, because the upper bits of the computed PAC context are different from the signing context. Likewise, if data belongs to the Compartment, but colored an inaccessible color, PAC authentication also fails, because the lower bits of the computed PAC context differ from the signing context. Only *valid* control and data flows are allowable inside the Clique, and that is enforced through requiring valid signatures before dereference. Table 2.2 presents a summary of compartmentalization operations, and the PAC context computations.

2.5.2 Developer Effort

Manual development is limited to specifying a compartmentalization policy, transferring dynamically allocated data to a Clique, and specifying which kernel data needs to be transferred to a Clique entry function before its invocation. While the annotations are lightweight (the largest single annotation we made is 74 lines to transfer data into a Clique), we have it as future work to automate these efforts.

The pointer validity checks and Compartment transitions are added via an LLVM [63] pass, which runs on annotated sources and skipped otherwise. Compiling the Linux kernel

Table 2.3. Summary of needed developer effort and automated instrumentation provided by the LLVM pass.

Operation	Developer	LLVM
Compartmentalization policy definition	✓	
Data ownership transfers from kernel to Compartment	✓	
Dynamic memory allocation transfers	✓	
Data access validity check insertions		✓
Valid Compartment transition check insertions		✓
Data ownership transfers to external Compartment		✓
Data ownership transfers from external Compartment		✓
Signature stripping in unprotected code		✓
Ensure all dynamic allocations are transferred		✓
Signing of global variable addresses		✓

using LLVM is supported, and no custom modifications to the LLVM source were needed. To establish a Clique, the LLVM pass places all similarly colored code and data into specially defined ELF sections, which the kernel module loader looks for when loading. When the compartmentalized LKM is first loaded, the kernel colors the code and data appropriately before any initialization code is executed. Once initialization code begins executing, the kernel performs its typical page-level permission enforcement in addition to the compartmentalization enforcement provided by HAKC. However, the page enforcement prevents changing the colors of code and read-only data, and for safety HAKC does not enable write permissions when changing colors. Therefore, the developer must be aware of these limitations when developing the access-control policy for Cliques.

As mentioned earlier, HAKC computes a PAC signing context using some compile-time information, specifically in which Clique and Compartment particular data and code belongs. Currently, the developer must annotate code and global data in the source to establish the Clique and Compartment membership. The annotations make the compartment membership permanent, since, for example, Compartment identifiers become encoded in instructions moving immediate values into registers. However, this is purely a performance optimization to avoid additional memory lookups, and HAKC can be extended to dynamically change compartmentalization policies at runtime.

In the simplest case where all functions and data defined in a source file belong to the same Compartment and Clique, the annotations are very lightweight; only a single addition of a macro defining the Compartment and Clique at needs to be added. In a planned future iteration, the LLVM pass will perform this operation for the user. If further partitions are required, the developer simply annotates the partitioned code or data, while the rest remains in the original partition. HAKC makes no attempt to optimize the developer-established partitions, The performance of the final system could be highly dependent upon the chosen partitioning, since Compartment transitions can be expensive due to the recoloring and pointer resigning process involved (see § 2.4). We leave it for future work to determine effective partitioning strategies, but HAKC supports any partitioning as long as the Cliques count in a Compartment does not exceed N_{tag}.

Finally, kernel module code is executed using function pointers registered to the kernel by the module during initialization. These function pointers represent the functionality the LKM implements. For example, a filesystem LKM implements a filesystem-specific `read` function, and registers that function with the kernel, which the kernel then invokes when reading the filesystem is required. Since all functions in a Clique expect all dereferenced pointers to be properly signed and colored, before executing compartmentalized code, the kernel must transfer the data to the target Clique. The developer must write a function to transfer input data, and provide that function to the kernel instead of the original function. Table 2.3 provides a summary of all developer efforts and LLVM pass actions.

2.5.3 Policy Creation

The modules that we compartmentalized in our proof-of-concept implementation imposed a very simple compartmentalization policy. Namely no more than two Cliques belong to a Compartment, and control flow outside a Compartment was limited only to the kernel. While such a policy is easy to define, and provides some protection, users will likely desire a more complex and automatically generated compartmentalization scheme. As mentioned in § 2.5.2, a more compartmentalized kernel makes attacks harder, but could affect performance due to more Compartment transitions that need to occur. Unfortunately, developing a

performance optimal compartmentalization policy is a similar to the Partition Problem, which is NP-Complete [64]. Therefore, we do not expect a general solution to optimal compartmentalization to be found. However, like solutions to the Partition Problem, we expect heuristics and dynamic analysis can provide good enough, if suboptimal, solutions for real world applications. Little work has been done on bare-metal commodity kernel compartmentalization, and we leave it for future work to develop strategies for automatic policy generation.

2.5.4 Optimizations

HAKC utilizes two major optimizations to achieve its low overhead, an interprocedural analysis that seeks to prove pointers have been validated by all caller functions, and an intraprocedural analysis for efficient data check placements. We detail each optimization here.

Interprocedural Optimization

Many functions in the Linux kernel are valid only for a single compilation unit, i.e., functions defined using the `static` keyword. Because no other code outside of its source file can rely on these static functions, for any such function F, we can determine the set of pointers dereferenced in F that all caller functions authenticate prior to calling F. Those pointers then do not need to be authenticated, and caller functions can provide the authenticated pointer values instead of the signed versions. To ensure that our analysis can determine the full set of dereferenced pointers, we schedule our LLVM pass late in the compilation process after function inlining. The analysis HAKC performs here is similar to live variable analysis, with every function maintaining a set of pointers, $P_{start,F}$, that are known to be authenticated at F's entry, as well as the set of pointers F authenticates, $P_{auth,F}$. P_{start} is initially the empty set for all functions, and let $P_F = P_{start,F} \cup P_{auth,F}$. At every call site to F, we check if each pointer argument, p, is in the caller function's P, and if the pointer argument is in all P, then $P_{start,F} = P_{start,F} \cup p$. This analysis repeats until a steady state is achieved, and no P_{start} changes. For the LKMs we compartmentalized, this analysis reduced the number

of data check insertions by 2%. The number of global and stack variables that needed to be signed, because their addresses are passed to other compartmentalization functions, is reduced by 8%. These reductions translate to 12% fewer data authentication checks, and 19% fewer transfers when performing the 100KB overhead experiment detailed in § 2.6.2.

Intraprocedural Analysis

A naïve approach to authenticating pointers would involve finding the set of dereferenced pointers, and placing the authentication of each such pointer in the first basic block of the function. However, some pointers are only dereferenced when specific conditions are satisfied, and thus authentication of those pointers needs to happen only when they will actually be dereferenced. Therefore, we place each pointer authentication only at the immediate dominator basic block of all said pointer uses. We also ensure that if a pointer use is within the same basic block as the authentication, then the authentication happens before the dereference. We only create one authentication per dereferenced pointer in a function, which can lead to some unnecessary overhead. If a pointer gets dereferenced in two basic blocks whose immediate dominator is different from either, then the authentication can happen without the pointer dereference taking place. This can occur, for example, when pointers are only dereferenced in error handling with a `goto` statement for final cleanup.

2.6 Evaluation

When evaluating HAKC, we wanted to answer the following research questions:

1. What is the overhead imposed by HAKC?
2. What is the overhead of using multiple Compartments in a single system?
3. Will users notice any difference in performance under real-world work loads?

Here we describe our experimental setup, as well as address our research questions through microbenchmark and simulated user browsing experiments. We performed all evaluations on a Raspberry Pi 4 8GB, and our kernel version was based off the Debian 5.10.24 source.

```
ldg xT, xN
  ldr x16, [xN]
  mov x17, #0xF0
  lsl x17, x17, #49
  and xT, x17, x17
```

```
stg xT, xN, imm
  ldr x16, =TAG_MEM
  mov x17, xT
  lsr x17, x17, #49
  str xT, [x16]
```

Figure 2.6. MTE Instruction Analogs

2.6.1 Instruction Analogs

As of June 2021, no hardware implementing MTE is available, and the most readily available hardware implementing PAC are Apple devices containing the A12 processor, and are unfortunately heavily locked down. Therefore, in line with the evaluation methodology of Liljestrand et al. [46], we ensure correct functionality using emulation, and measure overhead using instruction analogs in lieu of PAC and MTE instructions. An instruction analog is a series of instructions that consumes the same CPU cycles and the same memory footprint as the PAC/MTE instructions, but does not perform an actual check. Thus it can be used for accurate performance evaluation without a PAC/MTE-enabled processor. The PAC analogs are adapted from the PARTS system detailed in [46], and we detail the MTE analogs here.

Version 5.10.24 of the Linux kernel uses the load and store instructions for single or multiple tags, namely `ldg`, `stg`, `ldgm`, and `stgm` respectively. For the single tag instructions, the kernel only uses the post-index encoding. To simulate the `ldg` instruction, which writes the tag to bits 49–53 of an input register, we perform a load of the target address, and finally place a valid tag value in the appropriate register bits. To store a tag, we retrieve the tag from bits 49–53 of the pointer address, and write to a global variable. Multiple tag operations simply repeat these single tag operations. We took care to ensure that memory accesses occur at every MTE instruction to simulate a worse case memory tag access, but an actual implementation of MTE could include tag caching or other performance enhancements. Thus, we claim that our performance overhead is an estimation of *worst case* performance. The instruction substitutions are listed in Figure 2.6.

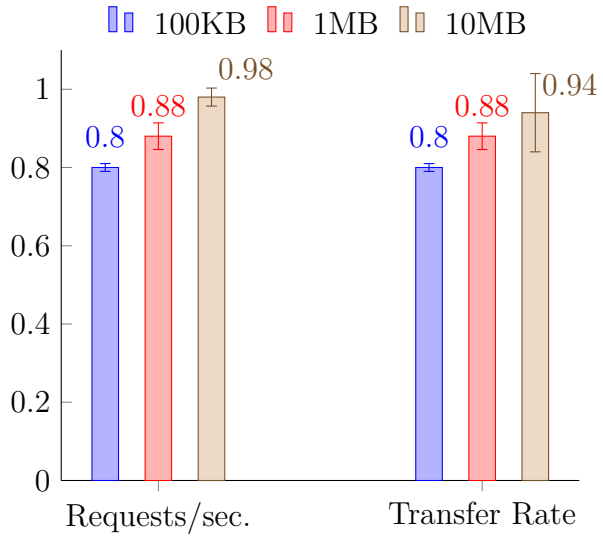


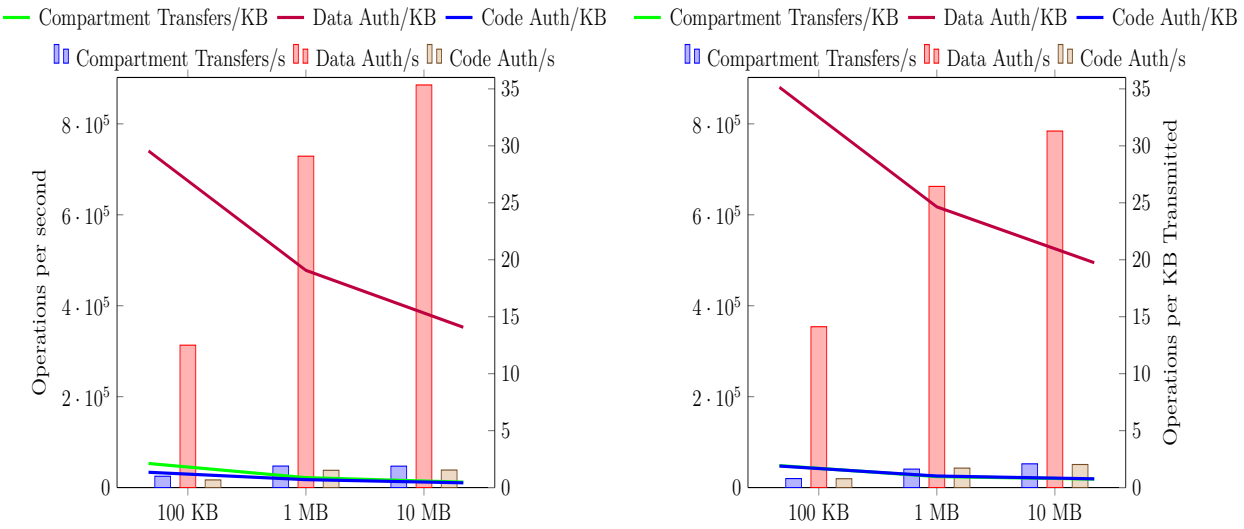
Figure 2.7. `ipv6.ko` overhead normalized to unmodified kernel when transferring various sized payloads.

2.6.2 Single Compartment Performance Overhead

To measure the compartmentalization overhead, we compartmentalized the `ipv6.ko` LKM into a single Compartment with two Cliques. Both Cliques were given access to each other's code and data. We used ApacheBench to retrieve a 100KB, 1MB, and 10MB file 1000 times from an unmodified Apache server running on the Raspberry Pi. We repeated each experiment 10 times, and recorded the reported requests/sec and transfer rate in MB/s. We measured all performance overhead relative to the unmodified kernel, and both kernels sharing the same user-space.

The overhead measurements for `ipv6.ko` are listed in Figure 2.7, normalized to the performance of the unmodified kernel. Overall, the performance of our `ipv6.ko` compartmentalized LKM is good compared to the baseline, with only a 20% reduction in both requests per second and transfer rate in the worst case.

When the transfer size is small, the establishment of the TCP connection imposes significant overhead relative to the actual transferring process. Once the TCP connection has been established, however, relatively few data checks need to be performed to transfer the payload. This explains the low 2%–4% overhead for the 10MB payload measurement; larger payloads spend less time establishing the TCP connection relative to the total transfer time.



(a) Only IPv6

(b) IPv6 and `nf_tables`

Figure 2.8. Average HAKC operations per second and per KB transmitted while running ApacheBench.

Figure 2.8a shows this behavior in the HAKC operations per kilobyte Apache sends. HAKC operations include the number of Compartment transitions, the number of data pointer authentications, and the number of code pointer authentications. While the number of operations per second either increase or remain constant, the number of operations per KB of transmitted data monotonically decreases with payload size.

2.6.3 Multiple Compartment System Overhead

`nf_tables.ko` implements a packet filtering mechanism within the Linux kernel. We compartmentalized this LKM by placing all code and data in a single Clique using a different color and Compartment from those used in the `ipv6.ko` LKM. The `ipv6.ko` and `nf_tables.ko` LKMs were not allowed to transition to each other directly. Since there is only a single Clique, no further compartmentalization policy needed to be specified.

To measure the overhead of using both LKMs on the same system, we defined a packet filter rule that drops packets with a source address from a specific IPv6 address. We then ran our microbenchmark detailed in § 2.6.2 using the unmodified kernel, the compartmentalized kernel with only the `ipv6.ko` LKM compartmentalized, and the compartmentalized kernel

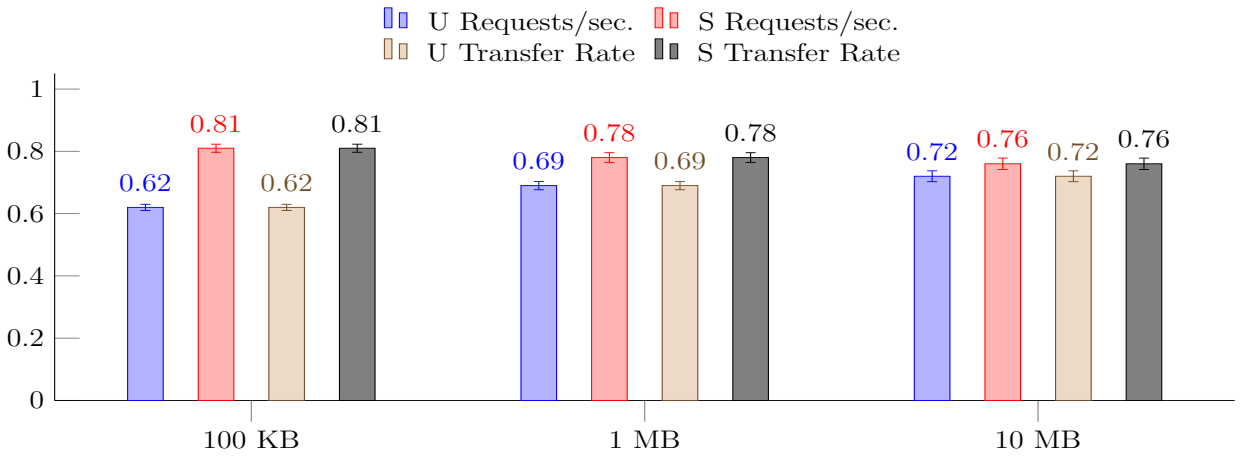


Figure 2.9. Overhead imposed when using multiple Compartments in a single system, normalized to the unmodified kernel (U) and single Compartment systems (S).

with both HAKC LKMs enabled. The results, normalized to the unmodified kernel (U) and `ipv6.ko`-only (S) kernel overheads, are listed in Figure 2.9.

The general trend regarding payload size and overhead shown in Figure 2.7 is again present for the overhead against the unmodified kernel. However, the performance relative to the single Compartment system degrades with payload size. The performance degradation comes from the additional Compartment transitions the kernel makes to perform both packet filtering and TCP functionality with every TCP ACK packet received. This behavior is shown in Figure 2.8b, with the higher number of data pointer authentications per kilobyte than with just IPv6 compartmentalized. Regardless, Figure 2.9 shows a linear growth of 14%–19% per compartment *when the compartments are related, but provide orthogonal functionality.* Compartmentalizing both the IPv6 and packet filtering represents a worst case for performance loss, since all HAKC operations for both LKMs will occur in tandem, and will thus be directly compounded. A better compartmentalization policy will likely amortize individual overheads to a lower total overhead, but we leave that evaluation for future work.

2.6.4 User Website Browsing

Using ApacheBench to measure raw performance does not necessarily provide a good indication of whether a user will notice any performance difference when using the compart-

Table 2.4. The measured time differences between the compartmentalized kernel of the lowest and highest standard deviations of unmodified kernel load times. Negative delta numbers indicate slower compartmentalized load time.

Website	Delta (s)	Stdev (s)
linkedin.com	-0.47	0.065
hdfcbank.com	-0.12	0.085
google.cn	-0.068	0.086
bing.com	-0.087	0.13
investing.com	38	62
okezone.com	-11	20
cnn.com	-9.8	15
yahoo.com	-4.9	15

mentalized kernel for everyday activities. For example, activities unrelated to the kernel networking stack, such as routing delays, website rendering, or advertisement negotiation, can add significant time to end-user web page loading. To answer Research Question 3, we want to measure any significant difference in IPv6 website loading time between using the unmodified kernel and our compartmentalized `ipv6.ko` LKM given these external factors.

To that end, we created a Selenium script that spawns a headless Firefox instance, and proceeds to play a specific YouTube video, and then visits the 50 most popular websites (as determined by the Alexa Top 1M) that advertise an IPv6 address in their DNS Authoritative Record (an `AAAA` entry). We disable all memory and disk cache use and enable IPv6 use in Firefox. Additionally, before retrieving each website, we delete all cookies, and perform a DNS query to ensure that ISP DNS entries are fresh. Afterwards, we measure the time the Selenium web driver takes to fully render the page, or the time the YouTube video takes to complete. To account for possible differences in advertisements, we retrieve each website using the unmodified kernel and compartmentalized kernel in turn before retrieving the next website. We repeated this experiment 5 times, with each retrieval separated by approximately 1 hour.

Overall, we measured the average load time of the compartmentalized kernel to be 1.19 ± 4.34 seconds slower than the unmodified kernel. Because the standard deviation of load time differences is much larger than the average, we conclude that the compartmentalized kernel

is not significantly different from the unmodified kernel, and that a user will not notice a difference using a compartmentalized kernel.

Despite our efforts to mitigate any possible difference between website retrievals, we did measure large differences in load times of some websites, on both large and short time retrieval spans. For example, `investing.com` would sometimes load in 4 seconds, and then after rebooting into a new kernel, the website would take 151 seconds. For this reason, we did not include `investing.com` in the average cited above. We were unable to determine any correlation between time of day or kernel type; the same website would be slow for the unmodified kernel at one time, and similarly slow for the compartmentalized kernel at another time, while the different kernels would statistically tie at every other time. The websites that exhibited the highest variance are those that serve dynamic content, such as `cnn.com`, while the lowest variance websites, such as `hdfcbank.com`, do not. We attribute the high variance to the underlying dynamic content generation on the server side, i.e., outside of our control.

Table 2.4 lists the websites with the smallest and largest unmodified kernel load time standard deviations, along with the measured time differences when using HAKC. In total, 20% (10/49) of the websites were measured to be faster using the compartmentalized kernel, and in all but one case, the load time delta was within 2 standard deviations (95% confidence). This provides further evidence that HAKC compartmentalization would go unnoticed by users in everyday usage.

2.6.5 Security Evaluation – CVE Case Studies

Here, we will provide a security evaluation on two real-world bugs, chosen to illustrate HAKC protection against bugs within and outside of compartmentalized code: CVE-2017-9074 [65] and CVE-2019-14815 [66]. CVE-2017-9074 is an internal IPv6 bug, while CVE-2019-14815 is an external bug in the Marvell Wifi driver. Of the 567 CVEs in our analysis set (see § 2.2.2), only 12 involved IPv6, demonstrating the importance of having compartments be hardened against external bugs, as most kernel bugs will be outside of a compartment.

```
int ip6_find_1stfragopt(struct sk_buff *skb, u8 **nexthdr)
{
    u16 offset = sizeof(struct ipv6hdr);
    struct ipv6_opt_hdr *exthdr =
          (struct ipv6_opt_hdr *)(ipv6_hdr(skb) + 1);
    unsigned int packet_len = skb_tail_pointer(skb) -
                              skb_network_header(skb);
    /* ... */
    while (offset + 1 <= packet_len) {
        struct ipv6_opt_hdr *exthdr;
        switch (**nexthdr) {
            /* ... */
        }
        offset += ipv6_optlen(exthdr);
        *nexthdr = &exthdr->nexthdr;
        exthdr = (struct ipv6_opt_hdr *)
             (skb_network_header(skb) + offset);
    }

    return offset;
}
```

Listing 2.4 CVE-2017-9074

CVE-2017-9074 (Listing 2.4) allows for reading memory outside the bounds of the intended object. The bug involves a missing check on `offset` against `packet_len` that ensures that the code is reading within the bounds of the socket buffer, `skb`. Through a series of system calls, a malicious user can craft an IPv6 packet that contains an invalid option, which causes `offset` to be much larger than the size of the allocated buffer for `skb`. `offset` is used to compute `*nexthdr`, which is read in the switch statement. This read is the out-of-bounds memory read.

HAKC prevents arbitrary out-of-bounds memory accesses like this, and instead limits the code's ability to only access the data explicitly allowable by the Clique `ip6_find_1stfragopt` belongs to. The large, corrupted `offset` value can place `exthdr` in one of several places: 1) a different Compartment and a different colored Clique; 2) a different Compartment but the same colored Clique; 3) the same Compartment and a different colored Clique; and 4) the same Compartment and the same Clique. In the first two situations, PAC authentication will fail because the computed PAC context will not match the PAC context used to sign `exthdr`. The third situation allows access only if the Clique is accessible according to the defined access-control policy, and the fourth situation will be allowed by HAKC.

```
256 void mwifiex_set_uap_rates(
        struct mwifiex_uap_bss_param *bss_cfg,
258     struct cfg80211_ap_settings *params) {
    struct ieee_types_header *rate_ie;
260 /* ... */

262 rate_ie = (void *)cfg80211_find_ie(WLAN_EID_SUPP_RATES, var_pos, len);
    if (rate_ie) {
264     memcpy(bss_cfg->rates, rate_ie + 1,
                rate_ie->len);
266     rate_len = rate_ie->len;
    }
268
    rate_ie = (void *)cfg80211_find_ie(
270             WLAN_EID_EXT_SUPP_RATES,
            params->beacon.tail,
272         params->beacon.tail_len);
    if (rate_ie)
274     memcpy(bss_cfg->rates + rate_len, rate_ie + 1,
                rate_ie->len);
276
    return;
278 }
```

Listing 2.5 CVE-2019-14815

To successfully perform this out-of-bounds read on HAKC-protected code, the attacker would have to construct `offset` such that the resultant pointer points to an accessible Clique, *and* contains the correct signature. The first condition already limits arbitrary accesses, and the second condition is computationally hard. This is how HAKC compartmentalizes code and data. The attacker is able to only access data allowed by the access-control policy, even in the presence of bugs, *and* the attacker must perform a computationally hard task to do so.

CVE-2019-14815 (Listing 2.5) is a bug in the Marvell Wifi driver that uses data from user-space in `memcpy` without checking the data length, leading to a heap overflow. Assume that the attacker uses this CVE from uncompartmentalized code to overwrite a pointer in compartmentalized code. The new pointer must again conform to all data access policies, and must contain a valid signature for the new pointer. Only if the new pointer is validly accessible and correctly signed, then the attack will succeed. However, as mentioned earlier, satisfying all the conditions is computationally hard.

Unfortunately, non-pointer compartmentalized data can be corrupted. However, this will likely only cause a denial of service, which, though severe, is considered less serious than

privilege escalation. One mitigation would be to utilize the "traditional" MTE, and store the color in the pointer along with the PAC signature. The MTE hardware can retrieve the color of accessed addresses, and check that value with the stored value, and throw a fault if they mismatch. The use of MTE and PAC in this way reduces the available signature bits by half, making brute force guessing of a signature easier.

2.7 Discussion and Threats to Validity

Here we discuss Hardware-Assisted Kernel Compartmentalization security and performance limitations.

2.7.1 Security Limitations

HAKC does not prevent all attacks. An attacker might find a valid control-flow path that adheres to all Clique and Compartment access policies, yet allows the corruption of data within a Clique. However, that corrupted data pointer cannot belong to some invalid Clique when dereferenced, and thus the damage the bug causes is contained to the compartmentalized code. Additionally, data provided by the kernel is assumed to be valid, which can lead to a confused deputy exploitation. Some bug in the kernel can allow invalid data to be signed and passed to compartmentalized code. Unfortunately, no practical solution to this problem, beyond formal verification [40], has been found. Instead, we envision a potential solution: formally verify the memory management and IPC code [67], and make all other functionalities HAKC-protected LKMs. Such a system could provide microkernel-like security, while keeping the robust functionality of existing kernels.

2.7.2 Performance Limitations

As indicated in § 2.6.3, LKMs that compute on the same data compound their overheads in the worse case. We have it as future work to evaluate the performance overhead of compartmentalizations that are largely unrelated. For example, we hypothesize that the overhead introduced by compartmentalizing `ipv6.ko` will not affect a compartmentalized Bluetooth LKM, and the overall system overhead will be the maximum overhead of either

compartmentalized LKM. We have it as future work to evaluate more compartmentalizations, and their effect on overall system overhead.

During the development of HAKC, we theorized strategies to reduce overhead of compartmentalization. For example, minimizing recolor operations by coloring all entry Cliques the same color might reduce overhead, since pointers only need to be resigned with the target ID_n. Additionally, static or dynamic analysis might indicate efficient compartmentalization policies. Existing tools, such as the Syzkaller [68] fuzzing engine or the KUnit unit testing framework, can provide insight into novel compartmentalization strategies. We also have it as future work to pursue interesting compartmentalization strategies, building on HAKC to allow empirical comparisons.

2.8 Related Work

Prior work includes isolation solutions that effect almost all parts of the computing stack, ranging from hardware extensions to novel user-space abstractions.

2.8.1 Isolation in Computer Systems

Kernel security is a long-standing and ongoing research topic. Prior work includes creating and improving isolation domains in both microkernels [69] and monolithic kernels [12], [36], [60], [70], [71]. Non-monolithic kernels, as well as some monolithic isolation methods, require significant kernel redesign, while HAKC is not as intrusive. Furthermore, HAKC allows for fine-grained isolation, unlike some of the methods listed above. There is also work regarding isolation in user-space [57], [72]–[78]. However, these techniques often rely on kernel abstractions, hence they are not applicable for kernel isolation, or would require the introduction of a trusted software layer beneath the kernel, i.e., hypervisor. Much of the work to handle privilege separation and isolation can significantly affect performance, hence works like Split Kernel [79] have been developed to select the level of protection and isolation for kernel functions based on if a trusted process is utilizing kernel functionality. HAKC is an *always on* solution that protects against exploits even from trusted processes.

Another approach for operating system isolation are library OSes [80]–[84], which restrict the operating system exposed to applications. Built on library OSes, multiple works [61], [85]–[90] have investigated unikernels — purpose-built kernels and user-spaces for a single application — and ways to create, improve, and use these minimalistic systems. Compared to HAKC, these approaches achieve isolation by separating kernel memory based on what each application needs. However, unlike the previous work, HAKC runs directly on bare metal, without any monitor reducing the trusted computing base. Furthermore, HAKC is much more flexible in defining different levels of granularity to allow for trade-offs between performance and security.

Finally, there have been research efforts in leveraging language properties to address memory related issues. Previous work uses Rust to implement operating systems [91]–[93] as well as unikernels [89], to utilize the language's type and memory safety to obtain isolation and increase security. While Rust prevents many memory related bugs, in order to prevent data-only attacks involving accessing valid, live memory areas, a compartmentalization system like HAKC is required.

2.8.2 Hardware Based Isolation

Intel's Memory Protection Keys (MPK) is an x86 extension that allows a process to partition memory into 16 domains. R/W privileges for each domain are then controlled by modifying a special key policy register, which is accessible from ring 3. User space access has motivated efforts to enforce isolation using MPK in a secure manner [58], [59], [94]–[97]. Certain MPK-based isolation schemes are vulnerable to attacks that leverage kernel system calls to subvert MPK permissions [98]. Unlike HAKC, MPK is designed to provide coarse-grained, page level isolation. Intel's Software Guard Extensions (SGX) [99] provide another avenue for compartmentalization, however, SGX is intended for user-space enclave-like protection against a malicious operating system. SGX has also been shown to induce significant overhead [100].

Other works have focused on using hardware to support *safe regions* — regions of memory only accessible by privileged instructions — but have only extended simulated hardware

and have focused on user-space applications [101], [102]. There are several works on using hardware tagging to support various compartmentalization and pointer bounds checking schemes [103]–[107], however most of these works are implemented on simulated architectures. One effort in particular, Mondrix [108], provides inter-modular Linux kernel compartmentalization using a 2-bit word granularity tagging extension [109]. Unlike HAKC, Mondrix is implemented in simulation, and requires a memory supervisor that monitors all kernel permission changes. Furthermore, Mondrix only implements inter-module isolation, whereas HAKC supports both inter-module and intra-module isolation. The Cheri project [110] provides architecture extensions to support pointer capabilities, which can be used to encapsulate memory. Cheri's fixed capability model provides less flexibility than PAC, where arbitrary information can be used as the context to sign pointers. Further, Cheri's focus on capabilities misses data-only attacks.

Arm TrustZone is a security feature on Armv8-A and Armv8-M [111] architectures that provides strict memory isolation between a privileged *secure world* and an unprivileged *normal world*. Lack of inter-process isolation between applications in the secure world as well as applications in the normal world have inspired isolation schemes that leverage the TrustZone ecosystem [35], [112]–[114]. Unlike HAKC, these systems focus on enforcing inter-process isolation and providing safe regions applications to store sensitive data.

Efforts have been made to provide isolation on embedded systems which lack an MMU to support full virtual addressing, leading to applications and kernels often times sharing a memory space [34], [115]–[118]. While these systems present interesting research questions, HAKC requires an MMU and advanced hardware features likely missing in embedded environments.

2.8.3 Arm PAC and MTE Extensions

Recent works have utilized Arm PAC to enforce control-flow integrity (CFI), spatial memory safety, and code pointer integrity (CPI). PACStack [119] is a CFI scheme that secures return addresses stored on the stack through a chain of hashing, where a hash for each return pointer is unique based on the current execution path of a program. PTAuth [47]

enforces temporal memory safety by storing a unique id at the base of data object, using the unique id as the PAC context during signing and authentication. PARTS [46] is an LLVM instrumentation framework that utilizes PAC to support a CPI scheme that is resistant to pointer-reuse attacks, and thwarts control-flow and data-oriented attacks. Compared to these schemes, HAKC can provide wider protection against many classes of attacks, and in some cases, like with PACStack, can be used in conjunction. HAKC is the first design to the best of our knowledge that utilize MTE-based isolation. However, designs have been proposed that would leverage MTE-like architectural features to improve the Clang AddressSanitizer [120].

2.8.4 Isolation with Hypervisors

Monolithic kernels such as Linux are known to be vulnerable to faulty or malicious subsystems, such as device drivers and network stacks. This issue has motivated researchers to leverage hypervisors and virtualization schemes to isolate kernel subsystems [60], [70], [121], [122]. One example of a hypervisor-based solution is VirtuOS [121]. VirtuOS isolates various Linux kernel subsystems by using the Xen hypervisor to create service domains. Although efforts are made to reduce domain communication overhead, the copying of data, file descriptor translation, and the migration of domain-specific information incur significant overhead. Another example is HUKO [123], also based on Xen, which isolates untrusted extensions. Both these schemes include the hypervisor in the TCB, while HAKC relies on hardware for enforcement. There are multiple known vulnerabilities in existing hypervisors, and although work has been done to address this [38], [124], [125], verifying hypervisor implementations is a difficult task. Unlike hypervisor based isolation schemes, which focus on isolating systems at the granularity of kernel modules and subsystems, HAKC is capable of compartmentalizing *bare-metal* LKMs at a finer granularity, including compartmentalizing subsystems within an LKM.

2.8.5 Memory Safety Mechanisms

While HAKC provides some memory safety protections, the so-called eternal war on memory [3] continues. Luckily, many memory safety mechanisms — in particular, stack memory

protections — are compatible with HAKC, and can be deployed alongside. DataGuard [126] improves upon the SafeStack [127] analysis to determine if stack-allocated objects (including return addresses) are vulnerable to spatial, temporal, or type-based attacks. Objects classified as safe are placed on a separate stack, and a runtime component limits access to the safe stack objects. Shadow stacks [128] employ a similar mechanism to separate function return addresses from the rest of the stack, and allows for the detection of return address corruption at low overhead. μRAI [129] is an embedded systems technique that reserves a register (which is never spilled) to store the correct runtime return address, and all possible return addresses are stored in RX memory. As embedded applications typically executes much smaller applications, μRAI is able to compute and store all possible return values. Finally, both tagged architectures [130] and PAC [131] have been used to protect the stack from attacks. Since HAKC uses both hardware technologies, the possibility to co-opt those techniques for stack protection exists.

3. FLEXIBLE COMPARTMENTS

3.1 Introduction

Imposing a least privilege policy on systems improves the overall security of the system even in the presence of bugs, because the possible damage is confined to the affected compartment. After compromising a compartment, the attacker is limited to the exposed API that compartment has access to, highly limiting their lateral movement. A true least privilege policy would break monolithic software into as many compartments as possible, and only allow highly restrictive interactions between small sets of compartments relative to the total compartment count. A least privilege policy would also enforce correct semantics in cross-compartment calls.

Given its importance and ubiquity, a least privilege policy for the Linux kernel is desired, but unfortunately, such a policy is difficult to define due to the Linux kernel size and scale. State-of-the-art efforts to compartmentalize kernels rely on heavy developer annotations [60], target small, niche systems such as IoT devices [34], [118], or only protect against control-flow attacks [33] but do not limit data access.

The reasons for a lack of compartmentalization are varied, but a major concern is the difficulty in determining what the compartmentalization policy *should* be. The size and complexity of modern software makes determining compartments, the privileges each compartment needs, and the interaction between compartments difficult, and too impractical for manual analysis. Another concern is the performance of a compartmentalized system. An efficient compartmentalization policy is one that places the most commonly interacting components together, minimizing the number of compartment transitions needed during execution. Managing performance and security tradeoffs is key to user adoption for compartmentalization. A one-size-fits-all scheme does not exist, because different users will have different security and performance needs. Any single policy would necessarily target one set of users over another. For example, a large healthcare company might be more concerned with securing their system against attack than a single user running a home server, and would be willing and able to accept higher performance costs for more security. Ideally, the user should define their desired security and performance needs, and a compartmentalization

policy is created to suit those needs. Inferring compartmentalization policies and optimizing for performance are two problems that remain unsolved.

In this chapter, we present a system for *Flex*ible *C*ompartmentalization policy generation, FlexC. FlexC automatically analyzes the source code of the Linux kernel to create a Call-and-Type Graph (CTG), a weighted, directed graph that summarizes the allowable data and code flow between compilation units. The static information can be augmented with dynamically measured data for potential performance gains by placing highly interactive code and data together in a compartment. Edge weights are computed using a ranking function, and a fitness function determines which compilation units should be merged in the final compartmentalization that is tailored to the user's security and performance needs.

The evaluation of FlexC is currently ongoing. We plan on generating differently compartmentalized kernels by varying both the number of compartments and the relative influence of dynamically measured data, and measuring how those factors influence the performance and security of the compartmentalization. Initial results from 6 compartmentalizations are presented.

In short, this chapter presents a method for automatically inferring compartmentalization policies that are tailored to users' needs. Additionally, we provide a methodology for measuring the effectiveness of generated compartmentalization policies in both performance and security gains.

3.2 Design

Here, we detail the design of FlexC, including the generation of the Call-and-Type Graph, which forms the foundation of FlexC, and how compartments are formed from the CTG.

3.2.1 Call-and-Type Graph

In order to automatically impose a least-privilege policy to compartmentalize a system, one must determine the privileges each compartment needs, and how each compartment interacts with other compartments. The data that a compartment accesses and the functions that it calls represents the compartment's required privileges, and thus both sets of

information must be captured. Through static analysis, we construct a *Call-and-Type Graph* (CTG) that encodes the total needed privileges for all possible compartments. The CTG is a directed graph consisting of compilation units as nodes, and edges between nodes indicating the possible privilege access needed of the tail node by the head node. The creation of the CTG involves a two stage analysis process, a local analysis on every compilation unit, and a global analysis that coalesces all privileges into the final CTG.

In the first stage, for every compilation unit, we gather information on the types used, the functions defined and directly called, the global variables defined and used, and which functions "escape" the compilation unit. By escape, we mean functions that are either callable from other functions outside the compilation unit (i.e., functions not declared `static`), and functions that are passed as function arguments, or stored as struct members. We focus on functions, as opposed to code bytes, because functions and function pointers are the main mechanism with which programming languages interact with executable data. Additionally, for every indirect function call, we capture the type of function that gets called. Most function pointer use in the kernel is for dynamic dispatch of functionality, and is retrieved by multiple kernel object pointer dereferences. As Lu, et al. [132], shows, matching the source of a function pointer with the functions that are written to the same struct member can reduce possible target set by up to 98%. Therefore, in an effort to reduce the size of the final graph, we also capture the source of function pointers used in indirect calls. See further discussion regarding indirect target elimination (and how this relates to security) in § 3.5.1. If any additional privilege information is available (e.g., dynamically measured interactions between two compilation units), the corresponding edge can be augmented with the data.

Once all the local analyses are performed, the Call-and-Type Graph is created by consuming all the information gathered in the first step. Every compilation unit C is compared against every other compilation unit, O. Privileges C needs from O (if any) are computed and stored on the edge connecting C and O. Privileges include global variable definitions, direct function calls, type of indirect function calls and their sources if it can be determined, and number of shared types. After every compilation unit is analyzed, the resulting graph is the CTG used for generating a compartmentalization policy.

```c
static int tegra186_gpio_remove(
    struct platform_driver *pdev)
{ /* ... */ }

static struct platform_driver
    tegra186_gpio_driver = {
     /* ... */
     .remove =
         tegra186_gpio_remove,
     /* ... */
};
static void module_init() {
   __platform_driver_register(
       &tegra186_gpio_driver,
       THIS_MODULE);
}
```

Listing 3.1 `gpio-tegra186.c`

```c
static int xlp_gpio_remove(
    struct platform_driver *pdev)
{ /* ... */ }

static struct platform_driver
    xlp_gpio_driver = {
     /* ... */
     .remove =
         xlp_gpio_remove,
     /* ... */
};
static void module_init() {
   __platform_driver_register(
       &xlp_gpio_driver,
       THIS_MODULE);
}
```

Listing 3.2 `gpio-xlp.c`

```c
static int platform_drv_remove(struct device *_dev) {
  struct platform_driver *drv =
      to_platform_driver(_dev->driver);
  /* ... */
  if (drv->remove) {
    ret = drv->remove(dev); // Could be either remove above
    /* ... */
  }
  /* ... */
}
int __platform_driver_register(struct platform_driver *drv,
                               struct module *owner)
{ /* Writes drv to struct dev kernel structure */ }
EXPORT_SYMBOL_GPL(__platform_driver_register);
```

Listing 3.3 `platform.c`

Figure 3.1. Kernel dynamic dispatch

Consider the code listed in Figure 3.1, in which two drivers implement a device removal function, and registers the removal function with the kernel. If a device is removed, the kernel will call the appropriate removal function through the registered function pointer. After FlexC analyzes the source code, the resultant CTG is shown in Figure 3.2. In this example, the two device removal functions escape their compilation units, since, despite being marked `static`, are written to an object that is passed as a function argument (the `struct platform_driver` global variable). When analyzing `platform.c`, FlexC finds the

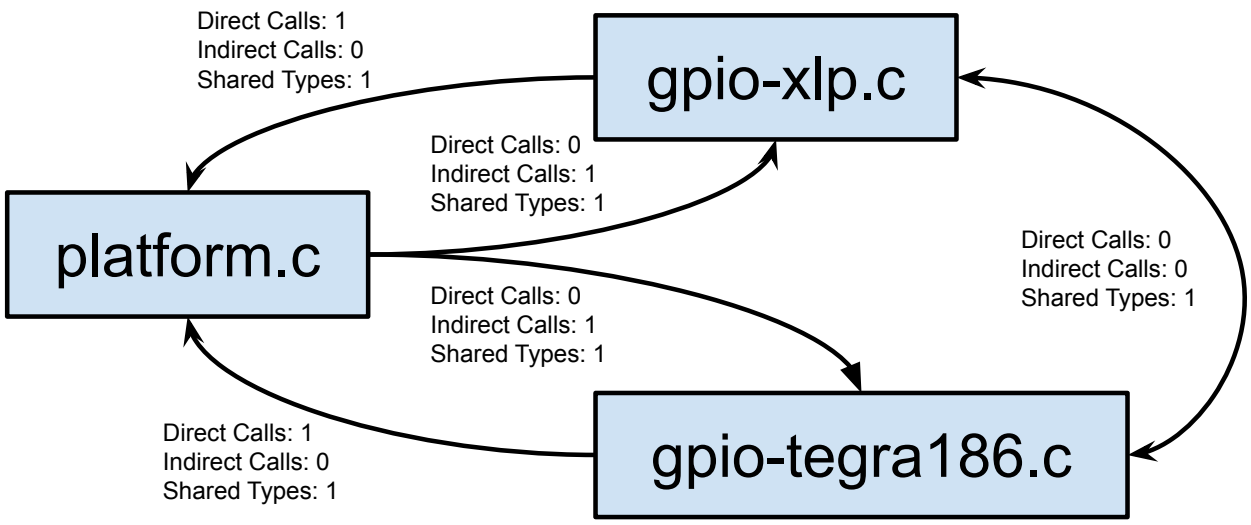


Figure 3.2. The CTG FlexC creates from the sources in Figure 3.1

indirect call, finds the source of the function pointer, and adds one to the indirect call count of all nodes that have a function written to the same type location. The drivers directly call a function in `platform.c`, so FlexC increments the direct call count for both edges by one. Finally, all three sources access the same data type, `struct platform_driver`, so FlexC increments the shared type count by one for all edges.

3.2.2 Compartmentalization Policy Generation

A compartmentalization policy can be seen as a partition of the CTG. The number of nodes in a partition allows for finer-grain compartmentalization policies, and thus stronger security guarantees, but also implies more work needed to ensure the compartmentalization policy is followed. Therefore, the number of compartments in the final compartmentalization policy broadly determines both the security and the performance of the final kernel.

A true least privilege policy would break the kernel into as many compartments as possible, while heavily restricting the allowable compartment interactions. In this sense, the CTG itself represents the strictest form of least privilege, and thus the most secure policy FlexC can achieve. However, since the CTG has the most number of compartments possible, it is also the least performant. By combining two nodes together, a more performant kernel can be made at the expense of (possibly minimally) expanded privileges. Combining two

nodes requires the definition of an edge ranking function, R, and a fitness function, F, that selects the two nodes to be merged based on the output of R. R can make use of any of the data stored with the edge, including any dynamic information. Thus, whether two nodes get merged is dependent on the weight given to any available dynamic data on the edge. For instance, one might opt to use only the dynamic information to create a very performant compartmentalization policy, and augment the compartmentalization with needed missing edges. We have implemented an R and an F for evaluation, but FlexC does not depend on this particular concretization, and future work can explore the impact of different choices in this design space.

FlexC allows the user to specify the number of compartments the final compartmentalization policy will contain. A new directed graph, G, containing the specified number of compartments with no edges is created, and all edges in the CTG are processed according to the output of F. The head and tail of the edge are placed in a compartment in G if they are not already in one, and if $R(head, tail) > 0$, then an edge between the two compartments in G is added. Once all the edges in the CTG are consumed, G is the compartmentalization policy. During compilation, our compiler pass (see § 3.3.1) reads in the policy, and automatically adds in the HAKC checks and transfers needed to enforce the compartments.

3.3 Implementation

In this section, we detail the implementation details involving changes to the existing HAKC instrumentation pass, the data that is used to construct the CTG, and how that data is used to partition the CTG into a final compartmentalization.

3.3.1 HAKC Instrumentation Changes

Previously, in order to specify a compartmentalization policy, the developer needed to manually annotate the source code to place code and data into a specific Clique. Such a mechanism does not allow enough flexibility, and is prone to error. We adapted the HAKC instrumentation pass (see § 2.5.2) to read in a compartmentalization policy during compilation, and add in the necessary changes based on the policy, and not source annotations.

Another challenge that hindered the performance of HAKC was the detection of per-CPU pointers, which are pointers to memory areas reserved to each CPU running on the system. These pointers, which are a form of lockless mutual exclusion, are stored as a base to which an integer is added to arrive at the final valid pointer. Detecting these pointers was relatively expensive compared to the rest of the HAKC data check, and had to be done at every HAKC data check. However, the kernel source utilizes an annotation to mark every per-CPU pointer, which unfortunately gets lost in the LLVM intermediate representation (IR). We modified `clang` to preserve this metadata in the IR, and our pass adds the per-CPU data check and data transfer wherever appropriate. A similar scenario exists for pointers from user space, and we further modified `clang` to preserve that information in the IR as well. However, for user pointers, we do not perform any validity checks, and we rely on the developers to properly and safely handle them.

We implemented other changes to the HAKC LLVM pass to facilitate FlexC. The original HAKC system forced the developer to write so-called transfer functions, which transferred data from the kernel to compartmentalized code before compartmentalized code can execute. This also does not scale when compartmentalizing large portions of the kernel, because thousands of transfer functions need to be written. Our new pass automatically creates transfer functions, and replaces the original function uses with the transfer function implementation. The kernel often uses function pointer value comparisons to determine control-flow. For example, the kernel prints a warning if the delayed work callback function in a `struct timer_list` is not the expected `delayed_work_timer_fn` function. Our pass also replaces function pointer comparisons with transfer functions where appropriate. Finally, our pass automatically transfers dynamically allocated data to the Clique that allocates it; something the previous pass required the developer to perform.

3.3.2 CTG Partitioning

In addition to the static information stored with the edges in the CTG, we also added dynamic information measured for the μScope evaluation [12]. This dynamic information consisted of the source and target of all function calls, as well as memory accesses (both

read and write) at the instruction level. We mapped the instructions to compartments in the CTG, and, for function calls, we augmented the relevant edge to include the number of function calls. For memory accesses, we added the number of accesses to the same memory location two nodes make to their mutual edge.

Our edge ranking function R computes a static weight, W_s, and dynamic weight, W_d, and uses a user-specified toggle to decide which final weight, W, to use. If the user wants to use dynamic weights, and $W_d == 0$ while $W_s > 0$ for some edge, R returns a W that is a non-zero value smaller than the smallest W_d in the CTG. This ensures that an edge is created in the final compartmentalization so no functionality is lost.

Our fitness function, F, for determining the order of edges to merge is a greedy algorithm that sorts the edges by their descending final weight W. There are four scenarios to consider when deciding which compartment to place the head and tail:

1. Neither the head nor the tail are in a compartment.
2. Exactly one is in a compartment.
3. The head and tail are in different compartments.
4. The head and tail are in the same compartment.

For the last scenario, no action needs to be done. The third scenario only requires adding an edge in G between the two compartments. For the second scenario, we place the uncompartmentalized node in a compartment selected at random that is already connected with the compartmentalized node. Finally, for the first scenario, we add both to the same randomly selected compartment.

3.3.3 Kernel Node

While most of the kernel consists of well-defined and self-contained systems, such as driver or protocol implementations, there also is core kernel functionality that is more difficult to compartmentalize. Examples include the memory management and interprocess communication, which are heavily vetted but widely used. Compartmentalizing important

code that receives a high level of scrutiny, but is used in many places throughout the kernel, provides little benefit while imposing high costs to maintain the compartmentalization. We, therefore, allow for the creation of a kernel compartment, where no HAKC checks are performed, however, transfers are still performed.

3.3.4 Metrics for Policy Evaluation

Compartmentalization is only effective if it results in measurable security gains. However, measuring security gains is an open area of discussion [2], [133]. We want FlexC to generate compartmentalization policies that significantly reduce the overprivilege of the whole system, and prevent attacks on common targets. Policies that are successful in both tasks are more valuable to users than policies that only do one, and both are needed to defend against current and future attacks. Attackers tend to focus on attacking crucial structures, e.g., `struct cred`, because permanent user privilege escalation is easily achieved if they are compromised. However, other, less well-known structures could be leveraged in attacks; we just do not know how yet. By lowering the overall privilege of the system, we can potentially mitigate future attacks, and generating policies that separate commonly attacked structures as much as possible mitigates current attack patterns.

To measure a compartmentalization policy's ability to reduce the total overprivilege of the system, and the separation of crucial structures, we propose the following two metrics: 1) normalized instruction privilege Hamming distance (NIPHD); and 2) Shortest Path to Crucial Object (SPCO). NIPHD provides the overall reduction of privilege in a compartmentalized system, while SPCO provides the compartmentalization policy's ability to mitigate known attacks.

The following formula computes NIPHD:

$$\sum_{i=0}^{N}\sum_{b=0}^{M}\frac{Hamming(i,b,R\vee W\vee X)}{3\cdot N\cdot M} \tag{3.1}$$

where N is the number of instructions, and M is the number of bytes of memory. NIPHD provides the amount of privilege reduction achieved by a compartmentalization compared to a completely permissive, monolithic system. For simplicity, we ignore page permissions, and

we make the assumption that code is always ∼W, and data is ∼X. Additionally, we measure NIPHD on the static kernel image, ignoring dynamic memory.

Computing SPCO involves gathering proofs-of-concept exploits to determine the locations at which exploits happen, and what objects the targets exploit. We call the exploit locations sources, and the target objects sinks. The SPCO is the shortest path in the compartmentalization policy from any source to any sink. Performing reachability analysis to critical objects or functions is a commonly used metric to demonstrate security gains [134].

3.4 Evaluation

Here we detail our experimental methodology to measure the overhead and security gains resulting from compartmentalization. The final evaluation is ongoing.

3.4.1 Effects of CTG Refinements

To evaluate the effects of compartmentalization, we will generate 6 different kernels, and run the full Linux Testing Project suite for each kernel. Each kernel will contain either 100, 1000, or the max number of compartments, and we will create static and dynamic versions of each compartment count. We will report the percent increase in execution time over the uncompartmentalized kernel as the overhead measurement. Additionally, in line with the HAKC evaluation, we also measure the number of HAKC data checks, code checks, and transfers performed per second.

3.4.2 Compartmentalization Security Evaluation

To compute the SPCO, we will gather proofs-of-concept exploits from the publicly available Exploit Database and from posts to the Linux Kernel Mailing List that target our kernel version, 5.10.24. Both SPCO and the NIPHD can be computed statically. We will present both metrics for every compartmentalization generated.

3.4.3 Initial Results

We currently have generated the static and dynamic compartmentalized kernels, and are currently measuring the overhead induced, as well as the effect the dynamic information has on performance. Our compartmentalized kernel consists of the code contained in the `net/`, `driver/`, `certs/`, `crypto/`, `virt/`, `security/`, `drivers/`, and `fs/fat/` subdirectories. These directories comprise 78% of the total lines of source compiled for the `AArch64` Linux kernel.

Figure 3.3 lists the percent increase in execution time for compartmentalized kernels, averaged across 5 executions of each listed test suite. The highest overhead comes from the net.features test suite, which takes the longest to run of all the evaluated benchmarks. Figure 3.4 explains the overhead source. Because FlexC compartmentalizes the majority of the kernel source, significantly more HAKC compartment transfers and authentications occur (approximately 30x and 7x respectively) than in the HAKC evaluation presented in § 2.6.2 and § 2.6.3. The optimizations and compiler changes detailed in § 3.3.1 — in particular, the removal of the per-CPU check performed at every data pointer authentication — can explain why the overall overhead does not similarly increase. However, further investigation is warranted, and we plan on performing the IPv6 experiments from the original HAKC evaluation with our improved compartmentalization instrumentation.

Only the net.features test suite shows any significant performance gain from the lowest compartment count kernels. The uncompartmentalized kernel takes, on average, 275 seconds to execute this benchmark, while the next closest benchmark, net.ipv6, takes only 9.27 seconds to complete on average. The longer testing time implies more of the kernel code is exercised during the net.features suite execution, which, in turn, results in potentially more compartment transfers. A compartmentalization policy that utilizes fewer compartments, however, would lower the probability that a code path leads to a compartment transfer. Figure 3.4 hints at this conclusion, as the 100 compartment kernels show a slightly lower compartment transfer rate, yet the rate of HAKC authentication checks remains constant regardless of compartment count. We will execute more of the Linux Testing Project suites to confirm this hypothesis.

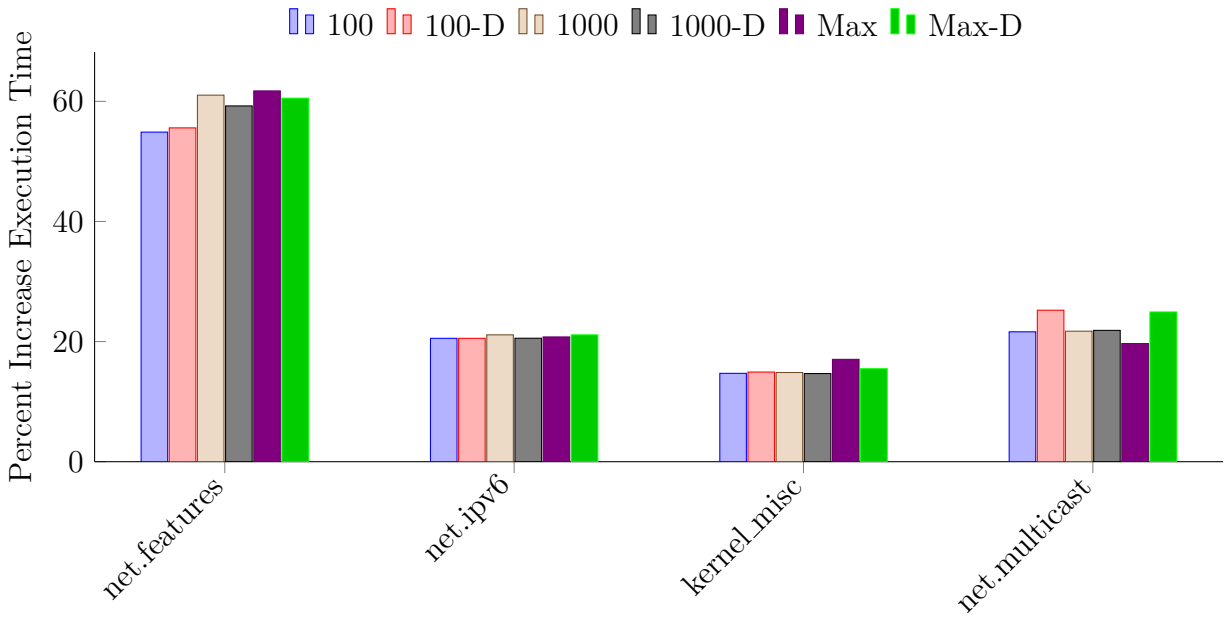


Figure 3.3. Average percent increase in test suite execution time over the uncompartmentalized kernel. D indicates dynamic data was used for the compartmentalization.

3.4.4 Dynamic Information Effects

The results from Figure 3.3 indicate that the dynamic information used to generate a compartmentalization policy does not have a significant effect on the performance of the resulting kernel. Therefore, we conclude that dynamic measurements are not needed when developing a compartmentalization policy. Considering the developer effort needed to dynamically measure kernel interactions, our finding that dynamic measurements do not contribute to the performance of a compartmentalized kernel is beneficial from an engineering perspective. Unique analyses do not need to be performed for every desired use case, and a single static analysis is sufficient for generating any number of compartmentalization policies.

3.5 Discussion

In this section, we discuss the elimination of indirect targets and alternative CTG partitioning schemes we experimented with.

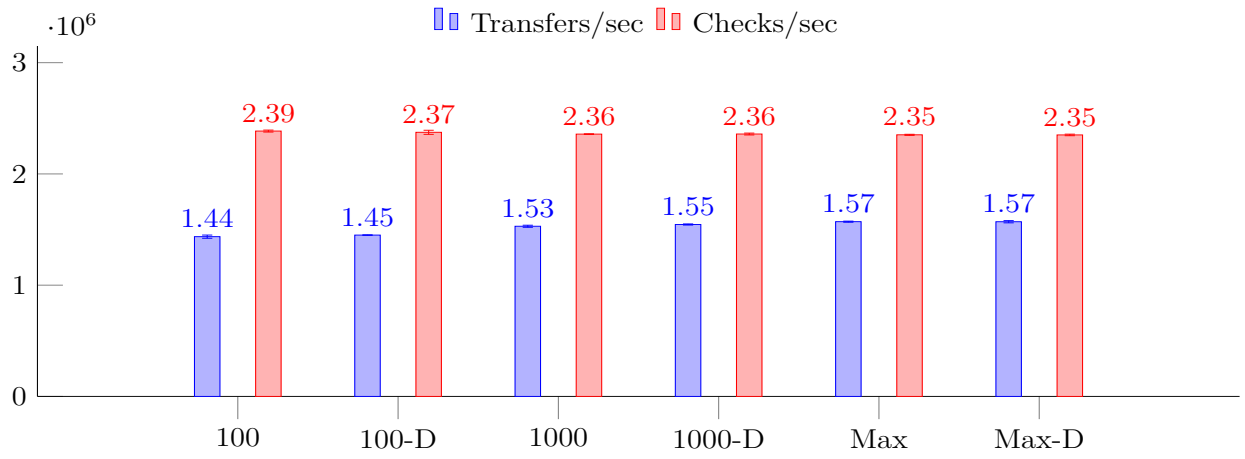


Figure 3.4. HAKC operations per second for different compartmentalizations when executing the net.features test suite. D indicates dynamic data was used in the compartmentalization.

3.5.1 Indirect Target Elimination

Previous work demonstrates that various methods achieve over 99% reduction in the size of indirect target sets [132], [135]. However, if an attacker can easily exploit the remaining target sets, the high reduction rate is meaningless. This is a problem for monolithic software, but much less so for compartmentalized code. Recall that compartments allow for the exploitation of bugs *inside* a compartment, but any external access is prevented by checks introduced by the compiler. An attacker conducting an attack involving indirect control-flow, e.g., a ROP-style attack, will still be confined to the compartment by the compartmentalization policy.

By reducing the size of the indirect target sets as discussed in § 3.2.1, we remove extraneous edges from the CTG, and, thus, remove even more options for an attacker to exploit. While an attack using over-approximated indirect target sets that obeys the compartmentalization policy is still possible, the attack is harder to perform, and the attack is useful only for the single victim; a different compartmentalization policy would invalidate the attack, because the attack is not allowable under the new policy. Additionally, preventing attacks that exploit bugs within a compartment are outside the scope of any least privilege policy.

3.5.2 Alternative CTG Partitions

In addition to our greedy partitioning algorithm (see § 3.3.2), we pursued two other possible partitioning algorithms. One algorithm used a multiple knapsack solver [136], and the other algorithm computed a min-k cut of the CTG. The multiple knapsack algorithm assigned each node a weight equal to the sum of its incident edge weights, and the used a multiple knapsack solver to place each node into a knapsack. Each knapsack in the solution would be a compartment in the new compartmentalization. The min-k cut algorithm computed the sets of nodes in the CTG that are disconnected if k edges are removed, starting with $k = 1$ and ending when all nodes are disconnected. All nodes that get disconnected at $k = n$ but are connected at $k = n - 1$ are placed in the same compartment in the final compartmentalization.

These two algorithms unfortunately do not create "good" compartmentalizations, in that the generated compartmentalizations do not place commonly interacting compartments together or effectively reduce over-privilege. The multiple knapsack algorithm places unrelated compilation units into the same compartment, because the solver simply fits any node into a knapsack. This is not ideal *for performance*, because nodes in the CTG with high edge weights will likely interact more, but will require more frequent expensive compartment transitions in G. This is because the two nodes will likely be placed in different knapsacks. The k-min cut algorithm is not increase *overall security* of the system. This is because there is a highly connected core of the Linux kernel, and then many weakly connected nodes that are disconnected at a low k. Essentially, small driver code was getting disconnected from the highly connected core when $k < 3$, and then creates no further compartments until $k > 30$. For these reasons, we decided to exclusively target the greedy algorithm, which results in relatively good compartmentalizations, as demonstrated in our evaluation.

3.6 Summary

Creating an efficient and effective compartmentalization policy is needed to enforce least privilege on computer systems. Manual efforts to create a compartmentalization policy are impractical, necessitating an automatic solution. This chapter proposes Flexible Compart-

ments as a method of automatically determining whole system compartmentalization policies tailored to desired performance and security requirements. FlexC constructs a Call-and-Type Graph of the system, which encodes the level of interaction between two source files. The CTG is then partitioned based on an edge ranking function and a fitness function, and the resulting graph is the compartmentalization policy.

The evaluation of FlexC is ongoing, but we plan on measuring several aspects of the compartmentalization policies it creates. We will measure the overhead imposed by compartmentalizing the majority of the Linux kernel using several configurations, created by varying the number of compartments and the level of dynamic data influence. Additionally, we will perform a two part security evaluation on the generated compartmentalization policies. First, we will measure the total reduction of privilege achieved by counting the allowable memory access all instructions have in both the compartmentalized and uncompartmentalized kernels. The ratio of compartmentalized privilege to uncompartmentalized privilege is the metric we will use to show the level of whole system privilege reduction, and compare the different compartmentalization policies FlexC generates. The second security evaluation is intended to show the effectiveness of FlexC at preventing exploits. We will gather a set of kernel proofs-of-concept exploits, and construct a set of source bugs and sink exploit targets. For each compartmentalization, we will determine how many sources are placed in different compartments from the sinks, preventing the exploit.

4. IOVEC FUNCTION IDENTIFICATION

4.1 Introduction

Semantic binary analysis—the act of determining a function's "purpose" within a binary—has applications in many research and engineering areas, such as plagiarism detection [137], code debloating [138], and malware analysis [139]–[142]. Patching third party libraries [143]–[146] requires determining the unpatched library version, *and* the full set of included functions, because developers will frequently distribute a custom-tailored version of a third party library that utilizes a subset of the possible functionality (e.g., only video decoding, and not encoding). [147] showed the first requirement is feasible to satisfy, but the second requirement is much less straightforward. Without source or an exact knowledge of how the library was generated, any user of a vulnerable library must either wait for the developer to fix the library, which can take on average over 500 days [147], or use semantic binary analysis to identify and locate vulnerable functions.

While source-based semantic inference work exists [26], [148]–[151], semantic binary analysis is a more difficult problem [14] due to the lack of information at the binary level. Manual semantic analysis does not scale to large binaries, necessitating an automated solution. So far, automated binary analysis [21], [152]–[156] measures *binary code properties* (e.g., order and type of instructions [154], memory locations accessed [23], [153], or control flow [22]), and approximates semantic similarity of functions based on the similarity of code. Although it is true that code similarity implies semantic similarity, the converse is not true—machine code may vary while still preserving semantics. We demonstrate that *program state modifications* serve as a better, more *stable semantic function identifier*. Program state change as a function identifier relies on the fact that semantic behavior is stable across compilations, environments, and implementations. Thus, program state change provides an ideal fingerprint, as it is impervious to compilation environment diversity or information loss. Code measurement approaches are susceptible to these complicating factors.

We present IOVec Function Identification (IOVFI), an approach to precise binary semantic analysis. Instead of relying on fragile function code properties, IOVFI abstracts functions into characteristic sets of inputs and corresponding program state changes. The core idea of

IOVFI is to observe and identify the *behavior or character* of functions instead of the underlying code, and then use the observed behavior as a unique function identifier. Our proof of concept IOVFI implementation automatically discovers a subset of a function's unique set of valid input program states and corresponding program state changes (referred to as Input/Output Vectors, or *IOVecs*). By observing data flow and program state transformations, IOVFI can classify functions, and, as a first-in-class feature, the IOVecs can transfer to different architectures with minimal effort.

We evaluate our prototype on accuracy amid varying compilation environments, a task existing works find difficult yet is crucial for binary patching and reverse engineering. We measure accuracy by identifying functions in the `coreutils-8.32` application suite, and find that IOVFI achieves a high .779 average accuracy across 8 different compilation environments. When identifying functions from differing compilation environments, IOVFI is 101% more accurate than the static *BinDiff 6* [155] framework, and 25%–53% more accurate than the dynamic *BLEX* [153] and *IMF-SIM* [154] frameworks. IOVFI achieves similar results to `asm2vec` [157] when compilation environments are similar, and significantly outperforms it for differing compilation environments.

We further demonstrate the generality of IOVecs by achieving similar accuracy when analyzing obfuscated binaries and `AArch64` binaries using unmodified `x64` IOVecs. We also demonstrate that IOVFI scales to large binaries by analyzing `libxml2`, `libpng`, and `libz`, which shows only a linear growth in training time relative to the number of functions in the binary. As an illustration of the utility of IOVFI, we perform a semantic analysis of 8 different versions of `libz`, and 6 different versions of `libpng`, and measure significant semantic differences which correspond to major changes to the underlying source. Finally, we use the `libpng` IOVecs to identify the versions distributed over the past 5 years of Ubuntu releases.

This paper provides the following contributions:

1. Design of IOVFI, a framework for semantic binary analysis that infers function semantics through program state changes;

2. A practical implementation of IOVFI that leverages coverage-guided, mutational grey-box fuzzing to automatically infer program states and input structure layouts for functions;

3. We show the effectiveness of IOVFI through a thorough evaluation on `coreutils-8.32`, obfuscated and cross-architecture binaries using unmodified IOVecs, and large shared libraries. We also perform a semantic analysis of 8 different versions of `libz`, and 6 different versions of `libpng`, and use the `libpng` training data to identify 5 years of Ubuntu distributed versions.

4.2 Challenges and Assumptions

Here, we outline challenges for semantic function identification, and our assumptions when designing IOVFI.

4.2.1 Semantic Function Analysis

Reverse engineering a binary is a tedious task. While initial extraction of binary code and determining the size and location of functions is non-trivial [25], [27], [156], [158]–[160], semantic identification is the hardest, most time-consuming part of reverse engineering. The largest impediment to semantically recognizing known functions is the large code diversity due to different compilation environments. Here, we refer to the compilation environment as the exact compiler and linker brand and version, optimization level, compile- and link-time flags, linker scripts, underlying source, and libraries used to generate a binary. Compilers attempt to create efficient, optimized code, and different compilers utilize different optimization sets. While compilers preserve the high level semantics expressed at the source level, the generated binary code is highly variable. For example, an analysis we performed on the `strlen` implementation in `musl` C library [161]—one of the simplest non-trivial functions in the C library—showed that simply changing the compiler could result in more than a 70% change in the disassembly. Optimizations, like dead code analysis and tail call insertions, also greatly affect the generated machine code. Even worse, custom function implementa-

tions (as opposed to the use of system-distributed libraries) will likely produce significantly different binaries.

However, regardless of compilation environment, the program state changes a function performs *must* remain stable for a binary to exhibit correct behavior. Barring any bug in the compiler implementation or inconsequential actions such as dead stores, the same source code should produce the same *semantic* behavior in the final application. If this was not the case, binaries would exhibit different, and likely incorrect, behavior in different builds. Therefore, measuring program state changes presents a viable method for semantic identification that does not rely on fragile measurements of code.

4.2.2 Assumptions

In line with existing semantic analysis tools, when designing IOVFI, we assumed the following:

1. Binary code is stripped, but not packed.
2. Binary code is generated from a high-level language with functions, and function boundaries are known.
3. Functions make state changes that are externally visible.
4. The binary follows a discernible and consistent Application Binary Interface (ABI).
5. Functions do not rely on undefined behavior.

When analyzing binaries, reverse engineers start with a stripped binary from which they infer its behavior. The analysts have no access to the underlying source, debugging information, symbol table, or any other human-identifiable information. We assume the same setting for IOVFI. Semantic analysis frameworks also make the assumption that all code is unpacked, and that the binary was generated from a high-level language with a notion of individual functions and a known ABI. The latter assumption precludes applications written wholly in assembly with no discernible functions, and, while packed code is another serious challenge in binary analysis [158], [159], [162], that topic is orthogonal to the analysis that

semantic analysis frameworks perform. Finally, as it is rare in practice and most likely a bug, no code may rely on undefined behavior to correctly function. The compiler is free to *use* undefined behavior for optimization purposes, but the original source should not rely on any specific compiler-based optimization utilizing undefined behavior for proper functionality. Note that functions which rely on randomness (e.g., cryptographic functions) are still valid; semantic analysis frameworks simply assume that function semantics do not change with the compiler.

4.3 IOVFI Design

IOVFI is a function identification framework, which infers program semantics by measuring the effects of execution. Instead of measuring code properties, it measures program state changes that result from executing a function with a specific initial program state. When a function executes, it does so with registers set to specific values, and an address space in a particular state, with virtual addresses mapped or unmapped to the process' address space, and mapped addresses holding concrete values. We refer to the immediate register values and address space state as the program state.

IOVFI performs its analysis by instantiating a specific program state before function execution, and then measures the program state post-execution. Measurable program state changes are writes to locations pointed to by pointers, data structures, and variables whose valid lifetimes do not end when the function returns. These types of changes necessarily must be made to registers or memory addresses outside the function's stack frame. This is because, once finished, any change would be overwritten by later instructions, and thus the program would have been more efficient had it not called the function at all. Functions that make only ephemeral changes are dead code, and the compiler will simply remove such code. Additionally, depending on optimization level, some program state operations, e.g., dead stores which write to addresses but are never read, can be removed from the final binary. We do not include such operations in the function's set of program state changes, but focus on *persistent* and *externally measurable* program state changes. We argue that

```
int my_div(int a, int b, int* c)
{   *c = a / b; return 0; }
```

Listing 4.1 An IOVec Motivating Example.

most user space functions conform to these standards, however, we discuss the limitations these standards impose in § 4.6.

We also consider the immediate return value of a function to be a measurable program state change, but exclude changes to general purpose registers (e.g., `rbx` on `x64`) and state registers (e.g., `rsp`). They are excluded because, for caller-saved general purpose registers, their values are immediately irrelevant upon function return, and state registers have no bearing on function semantics. Additionally, measurable program state changes preclude modifications to kernel state not reported to user space.

While executing, a valid program state for one function might cause another function to fault, and the same function can perform arbitrarily different actions based on the program state upon invocation. Therefore, a function implicitly defines the input program states it *accepts*—states where the function can run and return without triggering a fatal fault—and the corresponding output program states based upon these input states. We call these accepting input and corresponding output program states Input/Output Vectors, or *IOVecs*. A function A is said to accept an IOVec I if A accepts the input program state from I, and the resulting state from executing A matches the expected program state from I. If either of these conditions do not hold, then A rejects I. See § 4.3.3 for the discussion of matching program states. Assuming functions make changes to input program states which are measurable post-execution, we can reframe semantic function identification. Precisely identifying a function can be seen as identifying the *complete* set of IOVecs which a function accepts. We call that set the *characteristic IOVec set* (CIS).

Consider the toy example in Listing 4.1. An accepting input program state is one that has the first argument set to any integer, the second argument set to any integer except 0, and the third argument set to any properly mapped memory address. The memory location pointed to by `c` can initially have any value. The corresponding output program state has

the return value set to 0, and the memory location pointed to by `c` contains the value of `a/b`. An IOVec is a single concrete tuple of accepting input state and corresponding output state, and $CIS_{\texttt{my\_div}}$ is the full set of IOVecs `my_div` accepts. Note that only the first two arguments, the location pointed to by `c`, and the return value, are relevant, and that neither the full address space nor every register value are relevant.

Every function has a CIS, and we hypothesize that most functions have a unique (non-empty) CIS. A set of functions that share a CIS is called an *equivalence class*. For the sake of brevity, unless otherwise noted, when we refer to a function, we are actually referring to an equivalence class of functions with equal functionality.

In the general case, a function's CIS is unbounded. So for practical reasons, we attempt to find a subset of a function's CIS, which we call the *distinguishing characteristic IOVec set*, or $DCIS$. A $DCIS$ for function f, $DCIS_f$, consists entirely of IOVecs which f accepts, and only f accepts every member of $DCIS_f$. Another function, g, might accept a member of $DCIS_f$, but there is at least one IOVec $I \in DCIS_f$ which g does not accept. IOVFI is used to identify a function `foo` in a binary by providing `foo` with IOVecs $I_j \in DCIS_f$. If `foo` accepts *all* I_js, then we say that `foo` $\equiv f$.

IOVFI needs an oracle to provide IOVecs in order to semantically identify functions, but there is no definitive source of IOVecs. Our prototype was designed to be one such oracle, but other oracles can be devised. For example, IOVecs can be derived from unit tests or inferred from a specification. Symbolic execution [163], [164] or constraint tracking [165] could similarly be leveraged to create IOVecs.

The number of IOVecs IOVFI needs in order to be precise is highly dependent on the diversity and number of functions analyzed. The minimal theoretical number is equal to the number of functions being analyzed, because IOVFI needs at least one accepting IOVec to identify and distinguish a function. However, it is likely more IOVecs are needed to precisely distinguish functions, but the use of *differences* in semantic behavior for discrimination minimizes the number of required IOVecs. We currently focus only on accepting IOVecs for semantic identification, however using rejected IOVecs also provides valuable feedback. For example, if a function g rejects only 1 IOVec in $DCIS_f$, this can be a signal that g and f are semantically related.

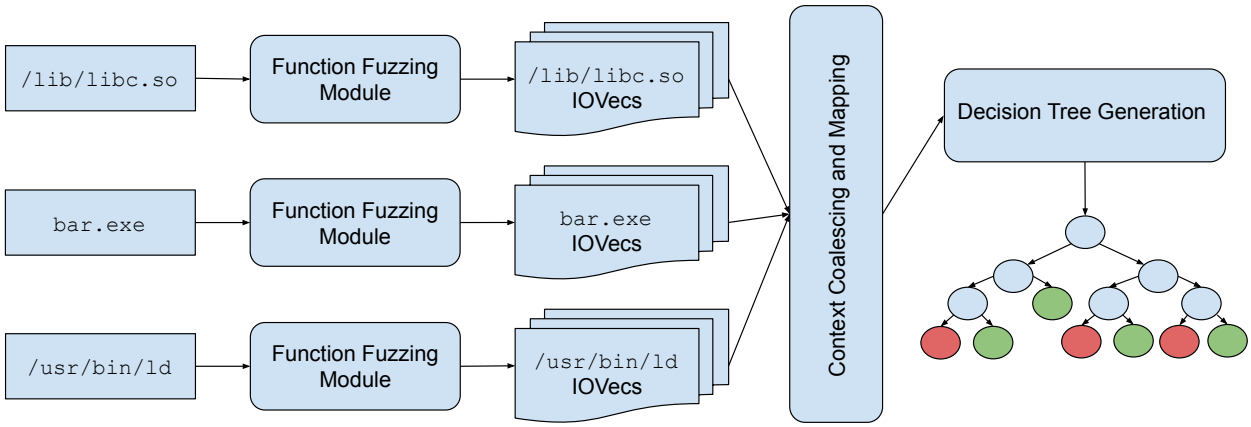


Figure 4.1. IOVFI Ahead-of-Time Learning Phase.

IOVFI performs its analysis in two phases: a coalescing phase and an identification phase. The coalescing phase, which only needs to be run once, is where functions are classified by IOVec acceptances and rejections, and ordered into a binary tree accordingly. The second phase is where unknown functions are semantically identified by providing the unknown functions with specific IOVecs from the binary tree, and traversing the tree according to IOVec acceptance.

Coalescing Phase

IOVFI starts its analysis by providing every function in its training set with every IOVec the oracle provides. This establishes a full ground truth of which IOVecs are accepted and rejected, ensuring that proper ordering can be achieved. Recall that an IOVec encodes both an input state and expected output state. When an IOVec is given to a function f, one of four results can occur:

1. The function receives a fatal signal (e.g., `SIGSEGV`), due to an improper input program state.

2. The function does not return before a specified timeout.

3. The function returns, but the final program state differs from the expected output program state.

4. The function returns, and the final program matches the expected output program state.

IOVecs that satisfy the last result are added to $DCIS_f$. As future work, we want to incorporate rejected IOVecs into the identification process, as rejected IOVecs classify the *rejected* semantics of this function.

The result of the coalescing is a proposed $DCIS$ for every function in the training set, and the $DCIS$ then fed to a decision tree generator. We use a decision tree generator (as opposed to another machine learning classifier) because, decision tree generators make classifications based on information gain, which is ideal for IOVec acceptance and rejection. The output decision tree contains IOVecs as interior nodes, and functions at leaves, and can be used for semantically identifying any number of functions later. As the tree is generated using *differences* in semantic behavior, it only grows linearly in the worst case. Every path from root to leaf encodes a minimal $DCIS$ needed to distinguish one function from every other in the tree. If the same path in the decision tree maps to more than one function, then a potential equivalence class exists in the binary. The functions in the leaf are those for which the generated $DCIS$ is insufficient to fully distinguish one function from another. This can be because the generated IOVecs cover the functionality poorly, or the functions are truly an equivalence class.

Identification Phase

Figure 4.2 shows the overview of the identification phase. To semantically identify functions, the analyst provides IOVFI with an unknown binary and the generated decision tree from the coalescing phase. Starting from the root of the decision tree, the IOVec is given to the unknown function. If the IOVec is accepted, the *true* branch in the decision tree is taken; otherwise, the *false* branch is taken. The unknown function is then tested against another IOVec depending on the path taken. When the path arrives at a leaf, the unknown function is tested against one more IOVec from the leaf function's $DCIS$ for confirmation. Again, if the IOVec is accepted, then the function is given the label of the function at the

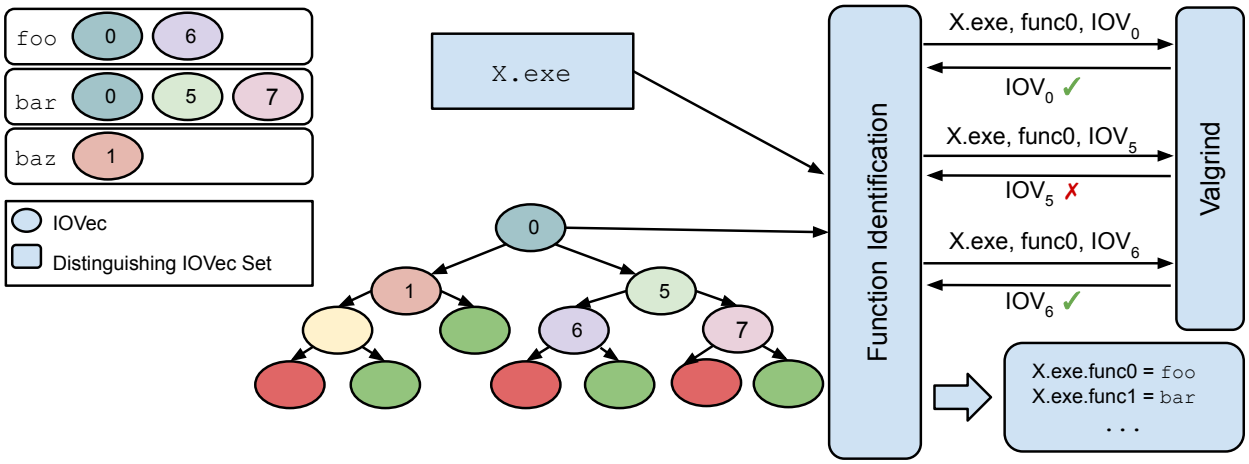


Figure 4.2. IOVFI Identification Phase. The ✓ and ✗ indicates that the IOVec was accepted and rejected respectively. Paths in the tree leading to green leaves indicate semantic equivalency in the unknown binary `X.exe` to a previously analyzed function (`foo`, `bar`, or `baz`), while paths leading to red leaves represent unseen/new behavior.

Table 4.1. Data stored in IOVecs.

IOVec Data	Use
Random seed	Program state initialization
Pointer input arguments	Program state initialization
Memory object information	Program state initialization
Code coverage	Fuzzer seed selection
Expected return value	Program state comparison
Expected memory state byte values	Program state comparison
Unique system calls	Program state comparison
Originating architecture	IOVec translation

leaf. If the unknown function gets to a leaf and remains unconfirmed, then the function is labeled as unknown.

The policy used for determining matching program states must remain constant for both phases. For IOVFI, we have implemented one such policy (see § 4.3.3), but others can be devised. The program state matching policy should take into consideration the memory model and features of the language in which the functions are written.

4.3.1 IOVec Discovery

IOVFI requires an oracle to generate IOVecs. Our prototype implements a coverage-guided mutational fuzzer [166]–[178] to infer IOVecs. Since we have no information about an unknown function's semantic behavior, the ideas behind feedback-guided mutational fuzzing are useful in discovering IOVecs. By rapidly feeding a function random inputs, and measuring the program state change post-execution, we can build a corpus of function identification data without any *a priori* knowledge. We chose fuzzing as our exploration strategy because fuzzing is optimized to maximize code coverage, leading to maximal program state change coverage. We do not need full path or code coverage to be accurate, only enough program state change coverage (i.e., data-flow coverage) to differentiate semantics. While limitations of fuzzing (e.g., passing complex data checks [179]) may limit the quality of the IOVecs, we observe that they are sufficient in practice. Our experimental results reinforce our main claim that program state change (however the IOVecs are generated) provides a more stable semantic identification fingerprint than code measurements.

Figure 4.1 shows the overall design of the first phase of IOVec discovery and coalescing. Our prototype supports analyzing any executable code, including shared libraries, but static libraries need to be included in either a shared library or executable. IOVFI requires neither the source nor any debug information; however, it does need boundary information of each function in an executable, or the exported symbol names in a shared library. Recent work shows that this information can be recovered even for stripped binaries [25], [27].

For each Function Under Test (FUT), our prototype fuzzes the input arguments and non-pointer memory object data if any have been deduced, and then begins executing the FUT with this randomized program state. If that program state is accepted, then the newly discovered IOVec is returned, and the resulting code coverage of the test is examined. If the IOVec produced new coverage, it is added to the FUT's $DCIS$, otherwise, it is discarded. Either way, the IOVec in the FUT's $DCIS$ that produced the most coverage (or a completely new, randomized IOVec in case the $DCIS$ is empty) is chosen as a seed for additional fuzzing. This process continues until the code coverage exceeds a user-defined threshold.

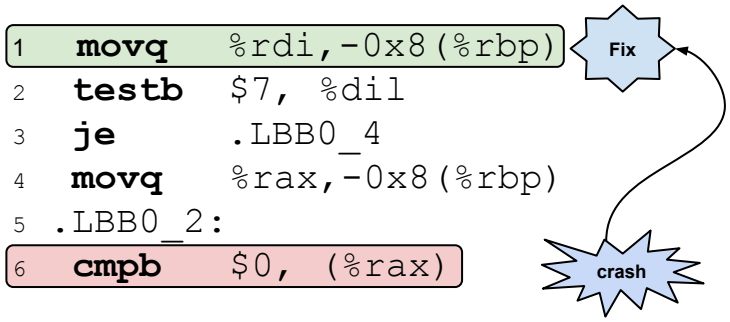


Figure 4.3. Backwards taint analysis to infer pointer arguments.

Table 4.2. Backwards Taint Propagation. t and u can be a register or memory address. T(x) taints x and R(x) removes taint from x. $\circ$ denotes any logic or arithmetic operator.

Policy	Instruction	t Taint?	u Taint?	Taint Policy
1	$t = u$	Yes	No	T(u); R(t)
2	$t = u$	No	Yes	
3	$t = u$	Yes	Yes	
4	$t = t \circ u$	Any	Any	

IOVFI stores the input program state and expected program state in an IOVec. Storing the entire address space is both a waste of storage and imprecise. Instead, IOVecs save the data listed in Table 4.1. Memory object information is the coarse-grained input layout and global memory objects inferred during the generation of the IOVec, and includes location, size, and pointer sub-member offsets. While generating IOVecs, our implementation uses code coverage to select an IOVec to mutate, so we include the instructions executed by the FUT when provided with the IOVec.

4.3.2 Pointer Derivation

A major challenge to generating high-quality IOVecs is the detection of pointers as input. As binaries contain no type information, determining if an input argument is a pointer is an ongoing research topic [180], [181]. Without recovering which arguments are pointers, determining a $DCIS$ is generally impossible, and only incomplete behavior will be captured.

A simple solution would replace an invalid address with a valid address before an illegal dereference occurs. While such a solution has been successfully used to solve other problems in binary analysis [15], [16], it would not work in IOVFI, because the underlying problem—semantically, an input is supposed to be a pointer when it is not—remains. IOVFI relies on capturing program state changes that arise from executing a function with a specific input program state. By replacing an illegal address *in situ*, the resulting output state does not necessarily arise from actions performed given the initial state, and an IOVec with an input state and an unrelated output state would be generated.

Consider the code in Figure 4.3, which is adapted from the `strlen` implementation in `musl`. The first pointer argument (passed in using register `rdi`) is stored on the stack (line 1). Later, that address is written to register `rax` (line 4), and then is dereferenced and compared with the null terminator (line 6). Our fuzzing strategy is unlikely to supply a valid address as input, and line 6 will cause a `SIGSEGV` signal to be issued.

The simple approach would replace the invalid address in `rax` with a valid address. If the function later returns with no other issue, then IOVFI would register `strlen` as accepting the input program state with `rdi` set to a random (non-pointer) value. This is incorrect, and during the identification phase, an implementation of `strlen` in an unknown binary would *not* accept the input program state. That `strlen` implementation would then be marked with an incorrect label.

The solution we propose is a backwards taint analysis inspired by Wang et al. [154], and illustrated in Figure 4.3. While generating IOVecs in its exploration phase, our prototype records immediate register values before every instruction executes, and, if a segmentation fault occurs (as in line 6 of Figure 4.3), we get the register containing the faulty address, which is the taint source. The saved register values are used to propagate the taint back to a root sink. The taint propagation policy is listed in Table 4.2. Starting from the last executed instruction, each instruction is parsed in reverse order until all instructions are iterated through. The root sink is the last tainted register or memory address after all instructions are processed. Our implementation utilizes the *Valgrind* framework [181], and its architecture independent intermediate representation, VEX. As VEX instructions represent a

single action, and we record all register values prior to executing a single machine instruction, we are able to precisely determine the root sink, and no false positives are possible.

After sink discovery, we search for previously allocated memory objects, and update the allocated bounds accordingly if an object is found. If no object is found near the faulting address, then a new memory object is built by allocating a fixed-size memory region, and records the current location and size of the object. We use this information for inferring new bounds and pointer sub-members if another segmentation fault occurs after execution restarts. Analysts can use the bounds information for more sophisticated analysis after decision tree generation. Once the object has been created or updated, the location is written to the sink, and begins executing the FUT from its beginning using the newly adjusted program state.

The backwards taint analysis restarts with every segmentation fault until the FUT successfully returns. When the FUT finally completes, we record the correctly initialized input program state, the corresponding output program state, and the coarse-grained object structure derived from the backwards taint analysis. IOVFI only tracks which memory areas are supposed to be pointers, and no other semantic meaning is given to memory regions containing non-pointer data. Further fuzzing iterations maintain the memory object structure, and only the non-pointer memory areas are fuzzed.

4.3.3 Matching Program States

IOVFI uses matching program states to differentiate and classify functions' semantics. Here, we present our definition of matching states used to identify C functions.

Recall that our notion of input program state includes memory objects for both global data as well as input arguments. Semantically similar functions modify memory objects in similar ways (if at all), so we capture the resulting memory state of allocated objects post-execution. Due to our fine-grained control over the memory state, any pointer value (either as an input argument or as a structure sub-member) is the same across executions. The allocated memory objects can be any arbitrary data structure, containing a mix of pointer and non-pointer data at various locations within the structure. Program states match when

non-pointer values in memory regions are byte-wise the same, and any pointers to sub-objects are located at the same offset from the object start. If there is a single mismatch in memory objects between two program states, then the states do not match.

Return values are also pertinent, but can be implementation dependent. We recognize two types of return values: pointers and non-pointers. Due to the lack of any type information in binaries, precisely determining if a return value is a pointer is challenging. We conservatively test if the return value maps to a readable region in memory, and if it does, we designate the return value as a pointer. If a return value is not readable in memory, then we consider it a non-pointer, and can represent functions that perform raw computations (e.g., `sin` or `toupper`), or adhere to a contract (e.g., `strcmp` which can return any value < 0, $= 0$, or > 0).

Finally, because system calls provide services that cannot be satisfied by user-space code and cannot be optimized out, semantically equivalent functions *must* invoke the same set of system calls. Order and number of system calls made, however, can differ among semantically equivalent functions (e.g., calling `read(fd, 1)` 4 times could be the same as calling `read(fd, 4)` once). Therefore, we include the set of unique system calls invoked while executing with the specific input as part of the IOVec. Semantically equivalent functions must invoke the same set of system calls, and can execute neither more nor fewer unique system calls.

For two program states to match, the values contained in return registers must match in the following ways. Return values must both be pointers or non-pointers. As we do not know the size of the underlying memory region, we do not check the underlying memory values if the return values are pointers; we simply say the return values match. Without more sophisticated analysis, this can be a source of inaccuracy. If the return values are non-pointers, they must be equal, or both must be positive or negative. If all input pointers (including pointers to all sub-objects) match, the return values match, and the same set of system calls are invoked, then the two program states match. As we do not perform any static analysis, `void` functions will also go through return value analysis, leading to another potential source of imprecision.

4.4 Evaluation

Our evaluation focuses on 64-bit System-V Linux binaries derived from C source code. We performed our evaluation using an Intel Core i7-6700K CPU, with 32 GB of RAM, running Ubuntu 16.04 LTS. We address the following research questions (RQ):

1. How accurate and scalable is IOVFI in identifying functions in binaries?
2. Is IOVFI truly resilient against compilation environment diversity?
3. Do IOVecs generated by IOVFI apply to other architectures?
4. Does IOVFI create meaningful equivalence classes?

Our results do in fact show that IOVFI is a feasible and accurate semantic function identifier. Additionally, our results show that IOVFI is largely unaffected by compilation environment changes, and that IOVFI can quickly identify previously analyzed functions. We show that IOVecs truly preserve semantics by achieving high accuracy when identifying functions in both purposefully obfuscated and `AArch64` binaries. Finally, our large-scale real-world application evaluation shows that IOVFI can scale to large, complex binaries.

4.4.1 Accuracy Experimental Setup

We selected `coreutils-8.32` for evaluation because the suite is a common evaluation metric in the literature, and used by both *BLEX* and *IMF-SIM* for their evaluation. To conduct our evaluation of IOVFI's accuracy, we selected `wc`, `realpath`, and `uniq`, which represent medium-sized applications using the default compilation environment. We compiled the set of applications using `gcc 7.5.0` [182] and `clang 6.0.0` [63], at `O0`–`O3` optimization levels. We then generated a decision tree (see § 4.3) for each application, for a total of 24 decision trees. The total amount of fuzzing time allocated for generating IOVecs was limited to 5 hours, after which the coalescing phase was allowed as much time as necessary. The coverage threshold to stop fuzzing a function was set at 80%. Only 19% of the classified functions hit that threshold during the exploration phase, and the average per-function coverage was 61%. While low coverage could miss important semantic features, Jiang et al. [151]

Table 4.3. Geometric mean F-Score (left) for `coreutils-8.32` per decision tree compilation environment (rows) across evaluation suite compilation environments (columns), and percent increase F-Score over *BinDiff 6* (right).

D-Tree \ Suite		O0				O1				O2				O3			
		LLVM		gcc		LLVM		gcc		LLVM		gcc		LLVM		gcc	
O0	LLVM	**.874**	***27***	.829	*49*	.728	*85*	.691	*66*	.702	*123*	.667	*98*	.694	*133*	.743	*139*
	gcc	.852	*56*	**.851**	***24***	.726	*97*	.685	*67*	.691	*142*	.655	*131*	.691	*141*	.736	*201*
O1	LLVM	.661	*74*	.692	*82*	**.891**	***30***	.636	*66*	.753	*79*	.690	*73*	.718	*73*	.671	*110*
	gcc	.848	*113*	.811	*91*	.815	*128*	**.852**	***34***	.808	*137*	.782	*104*	.804	*146*	.854	*146*
O2	LLVM	.723	*107*	.744	*121*	.836	*76*	.736	*89*	**.929**	***49***	.789	*107*	.916	*53*	.752	*91*
	gcc	.710	*85*	.757	*117*	.835	*117*	.718	*74*	.828	*129*	**.892**	***49***	.830	*138*	.799	*68*
O3	LLVM	.723	*110*	.742	*120*	.835	*77*	.735	*93*	.929	*54*	.798	*122*	**.926**	***51***	.760	*99*
	gcc	.849	*137*	.830	*173*	.825	*153*	.819	*128*	.822	*124*	.848	*78*	.820	*137*	**.932**	***53***

Table 4.4. `asm2vec` F-Scores (left), average similarity of true labels (middle), and average similarity of predicted label (right).

Training \ Test		O0						O3					
		LLVM			gcc			LLVM			gcc		
O0	LLVM	**.952**	**.973**	**.969**	.224	.537	.642	.0379	.270	.497	.0199	.333	.548
	gcc	.296	.596	.704	**.951**	**.966**	**.965**	.0379	.291	.500	.0467	.479	.636
O3	LLVM	.0656	-.218	.535	.0370	.283	.586	**.849**	**.955**	**.949**	.159	.612	.626
	gcc	.0519	-.0407	.453	.0108	.295	.565	.220	.381	.511	**.857**	**.920**	**.939**

found that in practice most functions are distinguishable using few executions. The accuracy in our evaluation further backs up this finding.

Each tree was used to identify functions in `du`, `dir`, `ls`, `ptx`, `sort`, `true`, `logname`, `whoami`, `uname`, and `dirname`, each also compiled using `gcc 7.5.0` and `clang 6.0.0` at `O0`–`O3` optimization levels, for a total of 80 binaries. These applications represent the 5 largest and smallest applications as determined by the default `coreutils` compilation environment. We used a subset of `coreutils` applications because an evaluation of one application requires $8 \cdot 24 = 192$ experiments. Evaluating all 100+ applications would therefore exceed $20,000$ experiments. Given that the applications share a lot of functionality, such an exhaustive evaluation is unnecessary. In order to establish ground truth, we compiled all binaries with debug symbols enabled. However, IOVFI does not use them for its analyses, and they were only used for determining accuracy after all analyses had completed. Unfortunately, some functions call `abort` or otherwise forcibly exit on invalid input, and thus our prototype in its current iteration could not properly analyze those functions.

We report the geometric mean F-Score (harmonic mean of precision and recall) across all compilation environments. In order to determine the correctness of a label, we performed a simple string comparison between the name of the FUT and the functions in the assigned equivalence class. If any matched, we record the function name as the assigned label, otherwise we use the name of the first function in the equivalence class as the assigned label. If a function is not matched to an equivalence class, we label the function as "*Unknown*". We then search for the function name among all the classified functions in the decision tree. The ground truth label is the function name if it appears in the classified function list, or "*Unknown*" otherwise. The classification labels and ground truth labels are then given to the `sklearn.metrics` Python module for F-Score calculation.

To evaluate against the most recent *BinDiff 6* (released in March 2020), we exported the needed input data using Ghidra [183] for each binary in every compilation environment, and performed pairwise analyses. The primary binaries were the decision tree binaries, and the rest of the binaries were the secondary binaries. Only the functions that our prototype classified were used for comparative accuracy measurements. We measured accuracy via a string comparison between matched function names, or with "*Unknown*" for secondary

functions that cannot be matched. The primary matched name was considered as ground truth for matched functions. For secondary unmatched functions, the function name was used as ground truth if it was present in the primary function list, while "*Unknown*" was used otherwise. Unfortunately, *BinDiff 6* only provides one function name for matched functions, so no further analysis could be performed.

`asm2vec` [157] is another state-of-the-art static similarity framework that uses natural language processing to infer a model of functions using known function disassembly as training input. Function similarity is performed by computing the cosine difference between two numerical vectors derived from the trained model, where one vector represents a known function, and one vector represents the FUT. The pair that yields the highest cosine difference is assigned equivalence. `asm2vec` will always return a similarity score (and a match), even when presented with a function the model has not seen. This feature presents a challenge in fairly evaluating IOVFI against `asm2vec`, because IOVFI is capable of declaring the untrained function as "*Unknown*," while `asm2vec` can only return a value between $[-1, 1]$[1].

To evaluate against `asm2vec`, we trained a separate model using the binary tree binaries, and used each model, along with the binary's functions, to identify functions in the test set. The function names of the top 2 results were compared with the FUT name, and the FUT name was used as the label if there was a match, or the top result label was used if there was no match. We used the top 2 results to fairly compare against the average equivalence class size that IOVFI differentiates (see § 4.4.3). Unfortunately, due to the long evaluation time needed for `asm2vec` (see the discussion in § 4.4), we could not evaluate it using the full training and test binaries. Instead, our `asm2vec` evaluation consists of the `O0` and `O3` `clang` and `gcc` decision tree binaries, and `true` and `logname` as test binaries, again only using the `O0` and `O3` versions. `true` and `logname` were chosen as representative of the small and large binaries in our evaluation set. We report the average F-Score `asm2vec` achieves while varying the compilation environment, along with the average true label cosine similarity, and the average predicted label similarity. The high F-Score that `asm2vec` achieves when the test and training binaries match compilation environments shows that, while imperfect,

[1]↑Technically, 0 could be construed as "*Unknown*", but utilizing it would be challenging, as most functions have *some* similarity with each other on the assembly level, and thus a 0 cosine similarity is rare.

this evaluation is reasonable given how `asm2vec` produces results. Due to the evaluation concerns with `asm2vec`, our evaluation focuses on *BinDiff 6*, as that system provides a more fair apples-to-apples comparison, despite its lower accuracy relative to `asm2vec`.

4.4.2 Accuracy Amid Environment Changes

Table 4.3 shows the geometric mean F-Score IOVFI achieved with decision trees from a specific compilation environment, along with the percent increase over the geometric mean F-Score achieved by *BinDiff 6* with the same environment. Each row reports the accuracy of all decision trees or primary applications from the specific compilation environment has when used to identify functions in binaries generated with a specific compilation environment (presented as the columns). The diagonal numbers (in bold) are, therefore, the accuracy rates when the decision trees or primary applications and evaluation suite match in both compiler and optimization level. They are unsurprisingly among the most accurate IOVFI and *BinDiff 6* achieved, and represent the data most reported by related work. *BinDiff 6* achieved an overall .402 ± .111 accuracy and standard deviation, and a diagonal accuracy of .642 ± .0387. Overall, we achieve a .779 ± .0777 accuracy rate, while the diagonal accuracy is .893 ± .0331, an improvement of 39%.

The off-diagonal numbers represent situations where training binaries differ from the evaluation binaries, and highlight the limitations of *BinDiff* and the strengths of IOVFI. As a static analysis framework, *BinDiff* performs its analysis using various graph comparison and hashing heuristics. While static analysis is significantly faster, those heuristics are based on fragile properties, as instructions and control flows change with compilation environments. Conversely, IOVFI relies on program state changes, which compilers guarantee will be stable across compilation environments and, as we will demonstrate, even across architectures. *BinDiff 6* achieves an average off-diagonal F-Score of .380 ± .0726 while our prototype achieves an off-diagonal .766 ± .0682 average F-Score, an improvement of 101%.

The generally high F-Scores across compilation environments indicate that our accuracy largely comes from IOVFI's ability to identify functions it has classified, and not from simply assigning an unknown classification to functions it has not identified. These results show

that IOVFI is accurate as a semantic function identifier, as well as resilient to compilation environments (RQ 1 and 2).

Table 4.4 shows `asm2vec` accuracy and similarity scores. We achieve similar F-Score values when the compilation environments match those of the training binaries, however IOVFI significantly outperforms `asm2vec` when the compilation environments differ. We attribute our use of F-Score as the accuracy metric, and the tight restrictions on the predictions that `asm2vec` produces for the difference between our results and the existing literature. Restricting the results to the two highest similarity functions, and incorporating both precision and recall into the accuracy metric makes achieving high accuracy a strictly more difficult task. The authors of `asm2vec` show that their system exhibits a inverse relation between precision and recall, which our results confirm. Conversely, IOVFI achieves high precision and recall.

The middle and right values of Table 4.4 list the average similarity scores measured for the true labels and for those that were picked as predictions respectively. In almost every case, the label similarity score is higher than the actual similarity score, indicating that incorrect functions are being measured as more similar than the correct function. Furthermore, the similarity scores measured for true labels from different compilation environments are significantly lower that those from matching compilation environments. For example, the true similarity score when using an `LLVM-O0` model to classify `LLVM-O0` binaries (0.973) is 45% higher than the scores measured while classifying `gcc-O0` binaries (0.537). As `asm2vec` claims, the further from 1.0 two vectors are, the less related the corresponding functions are, and, therefore, it was unexpected to measure such low (and even negative) average similarity among different compilation environments. The low similarity scores for the off-diagonal entries indicates that `asm2vec` is not well suited to analysis of binaries across varied compilation environments.

Additionally, we question the scalability of systems like `asm2vec` as semantic identifiers for large amounts of trained binaries. As noted earlier, our evaluation is only a subset of the full *BinDiff 6* evaluation because it could not complete in reasonable time. The main cause of the long processing time lies in the fact that the function vectors that `asm2vec` generate are independent entities that cannot be sorted in a meaningful way. Because of this independence, every unknown function must be tested against every classified function in

order to provide sound results. Conversely, IOVFI's ability to sort IOVecs into a binary tree creates an $\mathcal{O}(nlog(n))$ vs. $\mathcal{O}(n^2)$ classification disparity that results in significantly reduced binary classification time. We measured an average single vector pair comparison time to be small, taking only 0.12 ± 0.012 CPU seconds on average across $3,223,276$ vector comparisons, which is inline with the published literature. However, when all pairs of classified and unclassified functions must be compared, the total aggregate time to classify an unknown binary becomes large. IOVFI takes significantly longer to train, with `asm2vec` taking only 4 CPU minutes to train a model from one binary, versus hours for IOVFI. However, we emphasize that the training only needs to be done once, and afterwards classification with IOVFI is quick (see § 4.4.4).

Unfortunately, the two closest dynamic systems to IOVFI, *BLEX* [153] and *IMF-SIM* [154], are not available publicly. The *BLEX* authors supplied us their code, but it required significant engineering to execute with currently distributed Python modules. We invested two weeks of development and evaluation time. The accuracy we measured was much lower than the reported values, but this could be attributed to the required engineering changes or changes in the imported modules. The *IMF-SIM* authors remained unresponsive. We, therefore, base our comparison with these dynamic works on the published numbers, and call for open-sourcing of research prototypes. The *BLEX* authors report an average accuracy of .50–.64 across three compilers (they added Intel's `icc` compiler) and four optimization levels, and the *IMF-SIM* authors report an average accuracy of .57–.66 across three compilers and three optimization levels. Both systems attempt to build a classification vector from code measurements, and their lowest accuracies come from labeling functions in binaries from compilation environments different from their source models. IOVFI, in contrast, is accurate regardless of compilation environment, as evidenced by the off-diagonal numbers in Table 4.3. With a geometric mean accuracy of .766, our results show an average 25%–53% increase in accuracy in differing compilation environments over these works. The inaccuracy in *BLEX* and *IMF-SIM* arises from the fact that code measurements are not a true reflection of function semantics, but are instead one way to express function semantics from a large and diverse space of possible semantic expressions. The trained models they generate become inaccurate when presented differently optimized code, because they only capture a small

portion of the possible semantic expression space. IOVFI achieves its accuracy by actually measuring a function's semantics through program state change, and does not approximate function semantics through code measurements.

Despite its higher accuracy, IOVFI does have inaccuracy. We identify two major sources of inaccuracy: an overly strict program state comparison, and kernel state dependence leading to low-quality IOVecs.

Strict State Comparison

In § 4.3.3, we detailed our policy for comparing program states, which we use in lieu of code measurements for determining semantic similarity. We opted for a strict policy where both return values and allocated memory areas must match exactly in order for an IOVec to be accepted. However, at lower optimization levels, we might capture dead stores that are optimized out at higher optimization levels. For example, the `c_isprint` function, which returns a single byte, contains an additional `movzx` instruction in `O0` not present in any later optimization level. This instruction operates on the return register, which changes the higher order bits, while higher optimization levels simply write to the lowest byte in the return register without changing any further bit value. The write to the higher order bits is a dead store, since any caller will only ever read the lowest byte of the return register. However, we capture this behavior in an IOVec, and our strict return value comparison policy determines the return values to be different, leading to a mislabel. This is not a fundamental flaw with IOVFI, but an artifact of our program state matching policy. A different policy that more precisely compares program state could better account for inconsequential program state changes.

Kernel State Dependence

For simplicity, we designed IOVFI to assume nothing when generating IOVecs, and it always executes functions in isolation. However, there are functions (e.g., `close` and `munmap`) that depend on the results of previous functions in order for the input arguments to be semantically correct. For instance, `close` requires that the input integer be a valid open

Table 4.5. Geometric mean count of classified functions (N), average number of functions per equivalence class ($\overline{N}$) for all `coreutils-8.32` generated decision trees. The median equivalence class size is 1.00 for all decision trees.

	O0		O1		O2		O3	
	LLVM	gcc	LLVM	gcc	LLVM	gcc	LLVM	gcc
N	78	73	72	52	38	32	36	40
$\overline{N}$	1.76	1.85	1.82	1.68	1.78	1.47	1.69	1.69

file descriptor (as obtained from `open`), and any input that is *not* a valid file descriptor is semantically incorrect. Because we do not perform any initial setup to obtain semantically correct input values, any IOVec generated for these functions only exercise the error checking functionality, which is likely to be similar to many other functions. This has two negative effects: unrelated functions get grouped into an equivalence class, and unrelated FUTs can be assigned to this equivalence class simply because they share similar error handling behavior. This is, again, not a fundamental flaw in IOVFI, but instead is a result of our focus on user-space functions. We expect that our accuracy would improve significantly if we added some common environmental activities (e.g., opening file descriptors or memory mapping address spaces) to our IOVec design. We keep it as future work to incorporate application specific environmental setup to IOVFI.

4.4.3 Equivalence Class Distributions

Table 4.5 shows the geometric mean number of classified functions (N), and the average number of functions per equivalence class ($\overline{N}$). Ideally, $\overline{N}$ should be close to one, as most functions provide unique and singular functionality, and thus should be assigned as the sole member of an unique equivalence class. However, with the existence of wrapper functions, it is likely $\overline{N}$ will be higher. It nevertheless should be low, because one could trivially get high accuracy by grouping all functions into the same equivalence class. As Table 4.5 shows, we achieve a low $\overline{N}$ across our decision trees, which indicates that our fuzzing strategy is a generally sound technique for generating sufficiently distinctive IOVecs. Additionally, the equivalence class size distributions in Figure 4.4 show that we are creating hundreds of equivalence classes with one or two functions per equivalence class, which provides evidence

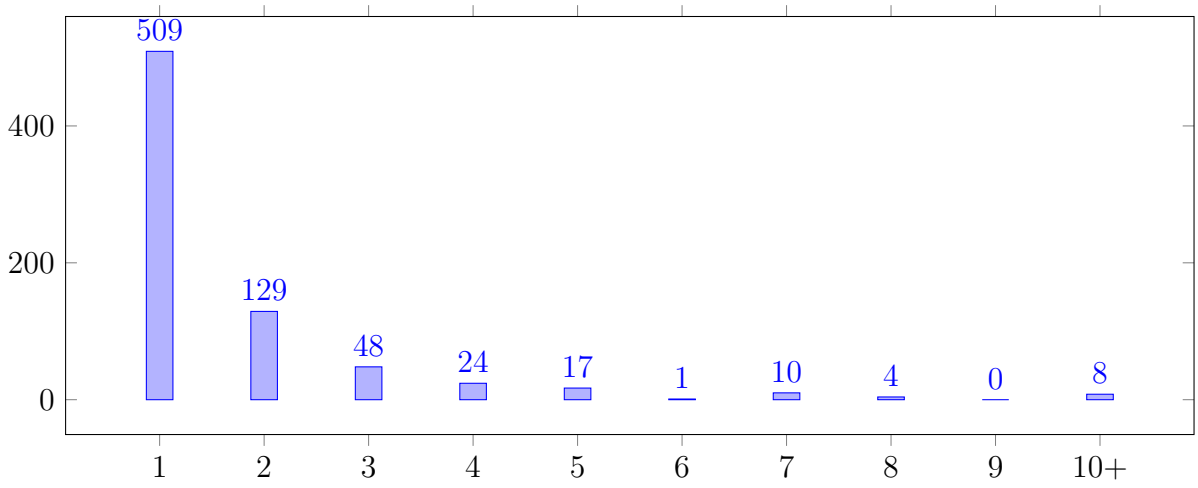


Figure 4.4. Distribution of all equivalence class sizes across all decision trees in the `coreutils-8.32` evaluation.

that we satisfy RQ 4. We, therefore, claim that our accuracy comes from IOVFI's ability to distinguish function semantics, and that our prototype does not simply group all functions into a few equivalence classes.

There are equivalence classes containing a large (10+) number of functions. These are cases where our fuzzing strategy was unable to trigger deep functionality, yet the classified functions share a common failure mode (e.g., return -1 for invalid input), or very similar functionality. For example, there is a 12 function sized equivalence class in the `realpath clang-O1` decision tree that contains 8 functions `strcaseeq[0-7]` that perform the same action with increasingly fewer input arguments. Improvements in related fuzzing work, especially works that improve deep code coverage [175], [179], will directly translate to an improvement of IOVec generation, and a reduction of the size of these equivalence classes.

4.4.4 Training and Labeling Time

IOVFI is scalable in both training time and storage requirements. On average, IOVFI takes 24.3 CPU hours to generate a decision tree, which includes generating IOVecs and the coalescing phase described in § 4.3. As stated before, however, this analysis only needs to be done one time. Once the decision tree is generated, semantic analysis is very quick, taking, on average, only 13.0 CPU minutes to classify a binary in the evaluation set. Additionally,

all operations in both of IOVFI's phases represent completely independent work loads, and as such are embarrassingly parallel. Therefore, execution time varies with the available hardware. Furthermore, the generated decision tree size is very small, with an geometric mean size of 855.9 KB. So, while IOVecs have no upper bound in their spatial size as they record the memory state of relevant inputs and their sub-members, in practice they are small.

BLEX reports 1,368 CPU hours for training, and 30 CPU minutes to classify a binary in `coreutils`. *IMF-SIM* takes 1,027 CPU hours for training, and 31 CPU minutes to classify a `coreutils` binary. Due to significant hardware differences between our respective experimental setups, and the lack of available source code for the related work, we cannot make any fair quantitative comparison. However, we believe that we are faster at semantic queries as we organize past analysis in a tree structure; *BLEX* and *IMF-SIM*, like `asm2vec`, must compare the feature vector they record with every past feature vector. Neither works report spatial size of their feature vectors, however *BLEX* and *IMF-SIM* restrict the number of instructions executed, which caps the size of their respective feature vectors.

4.5 Case Studies

We provide four case studies that demonstrate the effectiveness of our approach.

4.5.1 Accuracy Against Obfuscated Code

Malware authors will often employ code obfuscation to impede binary analysis [184], [185]. Code obfuscation attempts to hide semantic meaning through code transformations, such as adding unrelated control-flow or instruction substitution, while still preserving the intended function semantics. Code-based semantic analysis can be stymied when attempting to identify purposefully obfuscated code, because the resulting code is far from "normal," and thus hard to correlate with models derived from unobfuscated binaries. IOVFI, however, relies on *semantic* (rather than code) measurements guaranteed to be preserved by code obfuscators. Therefore, IOVFI should largely be unaffected by code obfuscation.

To test this hypothesis, we compiled our `coreutils` suite (`du`, `dir`, `ls`, `ptx`, `sort`, `true`, `logname`, `whoami`, `uname`, and `dirname`) using the LLVM-Obfuscator [186] at `O2`, enabling

Table 4.6. Obfuscated code accuracy comparison when bogus control-flow (bcf), control-flow flattening (fla), or instruction substitution (sub) is enabled for `coreutils-8.32`.

		IOVFI	*IMF-SIM*	% Difference
`gcc`	bcf	0.787	0.385	105
	fla	0.772	0.576	34.1
	sub	0.752	0.664	13.2
`LLVM`	bcf	0.806	0.513	57.1
	fla	0.795	0.649	22.5
	sub	0.813	0.779	4.30

separately the bogus control-flow (bcf), control-flow flattening (fla), and instruction substitution (sub) obfuscations. Following the experimental methodology of the *IMF-SIM* authors, we used the `OO` decision trees to measure semantic function identification accuracy in each of the three respective obfuscated binaries, using the same accuracy measurement metric described in § 4.4.1.

The results are listed in Table 4.6. We match or exceed the results achieved by *IMF-SIM*, with an average increase in accuracy of 39.3%. Our accuracy against obfuscated binaries, which closely matches our accuracy against unobfuscated binaries, provides evidence that IOVFI is unaffected by existing obfuscation techniques. Any inaccuracy when identifying functions in obfuscated binaries comes from the same sources as analyzing normal binaries, as discussed in § 4.4.2. Furthermore, these results also give evidence that RQ 2 is answered, as not only are the binaries purposefully obfuscated, but are also compiled using a much older version of LLVM than our evaluation version.

4.5.2 `AArch64` Evaluation

Function semantics are mainly determined by the high level source code, and remain largely constant across architectures. How the input state is established, and how the resulting program state is determined post-execution will change with architecture, but semantics do not. Therefore, an IOVec generated for one architecture is usable for another architecture, as long as there is a suitable IOVec translation between the two. In our implementation, we created a translation from `x64` IOVecs to `AArch64` IOVecs.

Table 4.7. F-Scores for identifying functions in `coreutils-gcc-O3 AArch64` binaries using decision trees generated from `x64 wc` (1), `realpath` (2), and `uniq` (3).

	O0		O1		O2		O3	
	LLVM	gcc	LLVM	gcc	LLVM	gcc	LLVM	gcc
1	.835	.805	.789	.840	.797	.803	.795	.860
2	.820	.803	.766	.794	.740	.761	.737	.842
3	.880	.866	.833	.791	.799	.849	.796	.877

Table 4.8. F-Scores identifying functions in `libz` (A), `libpng` (B), and `libxml2` (C) using a `clang-O0` decision tree. We did not evaluate against the `clang-O0` binary.

	O0		O1		O2		O3	
	LLVM	gcc	LLVM	gcc	LLVM	gcc	LLVM	gcc
A	-	.871	.717	.850	.759	.746	.765	.772
B	-	.781	.633	.695	.629	.642	.629	.639
C	-	.794	.699	.802	.701	.722	.700	.733

We evaluated IOVFI's cross-architecture accuracy by compiling the `du` and `dirname` (the largest and smallest binaries in our evaluation suite) on a Raspberry Pi 3 Model B Rev 1.2 running Ubuntu 20.04 using the ARM `gcc-9.3.0` compiler at `O3` optimization. We then used the unmodified decision trees generated for the evaluation described in § 4.4.1 to identify functions in the ARM binaries. The results are presented in Table 4.7, with each column listing the accuracy achieved using the `x64` decision tree generated with the enumerated compilation environment.

We achieve a mean F-Score of .811 across all the evaluated binaries, similar to our native geometric mean of .779. As our accuracy is largely unaffected by architecture, we strengthen our claim that IOVFI captures function *semantics*, and provide evidence that we answer RQ 3. Additionally, we also provide further evidence that we answer Research Question 1, as the `gcc` version used for this evaluation differs from the version used to generate the decision trees.

Table 4.9. Decision tree (N), average equivalence class sizes ($\overline{N}$), and CPU hours needed to generate the decision tree (T).

	`libz`	`libpng`	`libxml2`
N	126	390	2080
$\overline{N}$	2.47	2.48	2.44
T	17.0	25.4	158

4.5.3 Large Shared Libraries

Here, we demonstrate the scalability of IOVFI to larger, more complex binaries.

We chose `zlib`, `libpng`, and `libxml2` as a set of shared libraries that are ubiquitous and among the largest distributed with Ubuntu. We compiled each library using `gcc 7.5.0` and `clang 6.0.0` at `O0`–`O3` optimization levels, generated a decision tree for the `clang-O0` binary, and identified functions in the remaining binaries. Due to the larger size of the binaries involved, we allowed the fuzzing campaign to execute for 10 hours, and provided as much time as needed for coalescing. In order to handle the significant increase in functions, we used a machine with 45GB memory to generate the decision tree for `libxml2` (running Debian 9.3 on an Intel Xeon 3106). The machine listed in § 4.4.1 was used for all other evaluation tasks. The 50% increase in memory to process at least a 10x increase function count is a reasonable cost, and does not detract from our scalability claim.

Table 4.8 and Table 4.9 list the accuracy measured (using the same accuracy metric at in § 4.4.1), along with the number of functions classified (N), average number of functions per equivalence class ($\overline{N}$), and CPU time required to generate the decision tree (T). Our prototype achieves similar F-Scores as in our `coreutils` evaluation, while showing only a linear growth in T, demonstrating the accuracy and scalability of our approach (RQ 1). However, the number of functions per equivalence class is higher than our `coreutils` evaluation. This is a consequence of our simplistic coverage-guided fuzzer, as well as increased genuine similar functionality. For example, there are functions in `zlib` (e.g., `gzoffset` and `gzoffset64`) which only differ in the bit count of their input arguments, but otherwise perform the same action. There are also a large group of functions which first perform a sanity check on the input. The fuzzer did not create inputs to pass these checks, and the functions are grouped

into an equivalence class. Although inferring valid input is an ongoing research topic [174], [175], [179], both of these problems can be mitigated with a longer fuzzing campaign, a more sophisticated fuzzer, or through symbolic execution.

4.5.4 Semantic Differences and Versioning

Semantic function identification is required for binary patching if the compilation environment that created the binary is unknown. A binary might contain only a subset of the functions available in the source code, and identifying the full set of functions allows an engineer to generate a patch for any vulnerable function. IOVFI, since it is unaffected by compilation environment, is well suited to identify and locate functions within a binary for patch generation.

To demonstrate IOVFI's utility in binary patching, we analyzed the latest 8 versions of the `zlib` compression library, spanning `1.2.7` to `1.2.11`, as well as 6 versions of `libpng` identified in the LibRARIAN [147] Android app dataset. We kept the default compilation environment (`gcc O3`) constant across all versions, generated decision trees for each resulting shared library, and then used each tree to identify functions in every other version. As in the `coreutils` evaluation, if the FUT name appeared in the assigned equivalence class, then we considered the two versions of the FUT to be semantically equivalent, and otherwise, the semantics differed. Additionally, we manually verified a subset of mismatched functions for code changes resulting in semantic differences.

The differences in function semantics as a proportion of classified functions is listed in Figure 4.5, along with the number of additions and removals to source files between each pairwise version as reported by `git`. While some versions show sharp differences in semantics, (e.g., `zlib v1.2.9+` is significantly different from earlier versions), subtle semantic differences are also distinguished. As IOVFI does not rely on any information, besides function location within a binary, and the majority of functions within both shared libraries are not exported, we claim that IOVFI can uniquely identify exported and non-exported functions.

Key benefits of IOVFI are low analysis time to construct the dataset and very low matching time to query a function. The full semantic difference analysis of all 56 `zlib` version

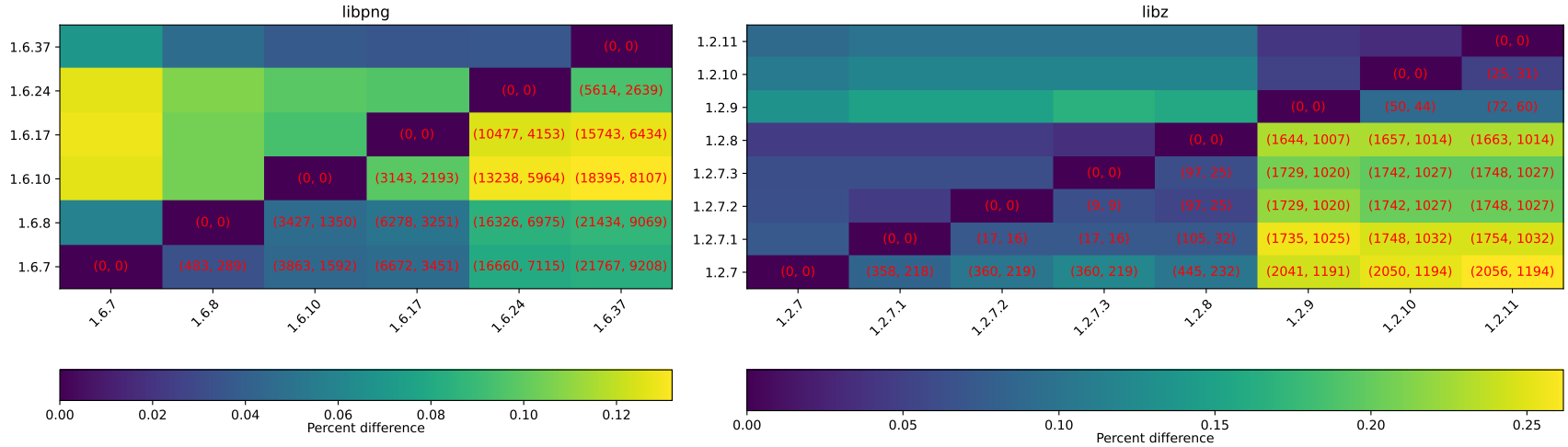


Figure 4.5. Semantic changes measured by function mismatches between IOVecs generated for a particular version (rows) and other versions (columns). Labels indicate the number of line additions and removals in source files between versions.

pairs took only 82 CPU minutes, while the 30 `libpng` comparisons only took 54 CPU minutes. Other approaches must compare each unknown function with every generated function model, creating an $\mathcal{O}(log(n))$ vs. $\mathcal{O}(n)$ search performance disparity between IOVFI and the current state-of-the-art.

Figure 4.5 shows that binary versions often have measurable semantic differences from each other, and thus those differences can serve as an identifying fingerprint for a particular version. When analyzing the exported functions of a shared library of an unknown version using decision trees generated from known library versions, the decision tree that produces the highest accuracy is likely to be the closest version to the unknown binary. LibRARIAN [147] performs this task statically (at a lost off precision), but IOVFI has the additional benefit of identifying non-exported symbols.

To test IOVFI as a shared library version identifier, we obtained the versions of `libpng` distributed for the past 5 years of Ubuntu releases, and analyzed each version with the `libpng` decision trees generated for the semantic difference evaluation. The decision tree with the highest accuracy was chosen as the candidate version, and we declared a successful match if that version is the closest to the actual version. The library versions include `1.6.37-3build3`, `1.6.37-3`, `1.6.37-2`, `1.6.37-1`, `1.6.36-6`, `1.6.34-2`, `1.6.34-1`, and `1.6.25-1`. In all but the `1.6.25-1` trial, IOVFI determined the correct version. For the unsuccessful trial, IOVFI selected the `1.6.37` decision tree, instead of the correct `1.6.24` decision tree.

4.6 Discussion

Here we provide discussion on the limitations of IOVFI, and on when a function is designated as unknown.

Limitations

We have identified a few sets of functions that IOVFI is unlikely to classify or identify correctly. These functions are highly dependent upon the system environment and execution context while generating IOVecs, as well as during the identification phase. Functions like

`getcwd` or `getuid`, which return the current working directory and the user ID respectively, depend on the filesystem, current user, and kernel state. As these factors differ between runs or are non-deterministic, they violate our fundamental assumption—semantically similar functions change their program state in similar ways given a specific input program state. To address this limitation, IOVFI could model the system state in addition to the process state.

Another set of functions IOVFI struggles with depend on an initial seed being set beforehand. Examples of these functions include `rand` and `time`. As we execute functions without any knowledge about their behavior, we cannot provide the seed beforehand as it is difficult to distinguish a seed value from other global variables. Even if we determine a location of the seed, knowledge of proper API usage (e.g., calling `srand` before `rand`) is needed to correctly use these functions. Discerning correct API usage is an active research area [187], and improvements in this area will directly translate to improvements in IOVFI.

Soundness of IOVFI

When semantic equivalence is determined between two functions, that equivalence is only extended as far as the IOVecs tested along the decision tree path. It is possible that IOVFI establishes an incorrect semantic equivalence between a previously analyzed function `f`, and a new unseen function `g`, if 1) `g` accepts all of `f`'s IOVecs, plus additional IOVecs; and 2) any additionally accepted IOVec is not along the path to `f` in the decision tree. This means that IOVFI is not a sound technique. However, as our equivalence class distributions results show, in practice IOVFI is accurate for most functions, even when functions are similar, as with `strcpy` and `strncpy`. In real world code, most functions have little overlapping functionality, which makes IOVFI a practical tool for semantic identification. We leave it as future work to incorporate code coverage into the semantic similarity analysis, which could produce more accurate classifications through the enforcement of a coverage policy as a condition for semantic equivalence.

System Calls

Currently, IOVFI only records which system calls are made during the execution of a FUT, and no further information is captured, and no further modeling is performed. This design choice is purposefully incomplete to avoid expensive operating system state replication. Most functions make no system calls; less than 3% of functions in `libxml2` call `read`, `write`, `open`, `close`, or their `FILE*` equivalents, for example. Of the functions that do make a system call, we assumed that they ignore the exact state of the operating system, and rely solely on the result of the system call. Our high accuracy justifies this assumption, and while state modeling could improve coverage, we believe that only marginal gains would result.

Unknown Functions

If a function is encountered that accepts no known $DCIS$, IOVFI will mark this function as unknown. When a function is marked as unknown, it can mean one of two things depending on the number of accepted IOVecs. If the unknown function *never* accepts an IOVec, then it implements wholly unknown functionality, and should be a main focus for analysts. Otherwise, if the function accepts some IOVecs, then it shares some functionality with the functions whose $DCIS$ includes the accepted IOVecs. The utility analysts might gain from this information varies with the number of IOVecs accepted. Many IOVecs rejected with a few IOVec acceptances is likely a common failure mode present in many functions, e.g., returning -1 on invalid input. If many IOVecs in a $DCIS$ are accepted, then the unknown function is likely similar to the corresponding function, indicating, e.g., a different version.

4.7 Future Work

IOVFI utilizes a mutational fuzzer to generate a function's $DCIS$. By incorporating more sophisticated fuzzing and binary instrumentation techniques [188], [189], it is possible to generate a $DCIS$ that provides close to 100% edge or code coverage of a function. Later, if that function is identified in a new binary, then any deviation in code coverage when given

the full coverage $DCIS$ would indicate the presence or lack of functionality in the FUT. This could be helpful in exploit generation, or code version identification [142].

A challenging aspect of reverse engineering is the detection of cryptographic functions in a binary. They are difficult to identify, because they are often implemented using architecture-specific assembly for optimization purposes, make extensive use of randomness, and rely heavily on correct state and input. These are situations for which IOVFI is particularly well-suited, and it would be worthwhile to investigate how far we can advance automated analysis on this most difficult class of functions. IOVecs, as an extension of the captured state, could record the random values returned by RNGs. In the coalescing and identification phases, calls to RNGs could be intercepted, and the recorded random value could be returned.

4.8 Related Work

Similarity analysis is an active area of research [18]–[20], [24], [157], [190]–[196]. Jiang et al. [151] first proposed using randomized testing in function similarity analysis, drawing inspiration from polynomial identity testing. Their EqMiner system, which requires source code, finds syntactically different yet semantically similar code fragments in large (100+ MLOC) code bases. A direct comparison between IOVFI and EqMiner is unfortunately challenging. Besides requiring source, which IOVFI does not use, the correctness metrics and similarity assertions used between the two systems are different. For example, EqMiner will declare two functions similar if they add two integers, irrespective of whether the integers are part of a struct or raw data types. IOVFI will mark the two functions as different, because of the different semantic uses in the whole binary. EqMiner defines similarity orthogonally to the data format while IOVFI uses IOVecs as the fundamental distinguishing factor. Both answers are correct for their respective use cases, but are incompatible when trying to evaluate one system over the other.

Current state-of-the-art binary analysis tools all rely on code measurements. *BLEX* [153] extracts feature vectors of function code, such as values read and written to the stack and heap, by guaranteeing that every instruction is executed. The authors also implemented a search engine with their system similar to IOVFI. Wang, et al. [154] perform code similarity

analysis using a system called *IMF-SIM*. *IMF-SIM* uses an in-memory fuzzer to measure the same metrics as *BLEX*, instead of forcing execution to start at unexecuted instructions. As stated in our evaluation, these works still struggle with differing compilation environments, while IOVFI has consistently high accuracy irrespective of compilation environment. Both works focus on measuring code properties, which change with different compilation environments. IOVFI, in contrast, uses IOVecs, which are independent of code, and encodes differing semantics in a binary decision tree.

Pewny, et al. [22] compute a signature of a bug, and search for that signature in other (possibly different ISA) binaries. The signature involves computing inputs and corresponding outputs to basic blocks in functions' CFGs through dynamic instrumentation similar to IOVFI. While the authors admit that semantic function identification is not their expected use case, their system can be used as such by supplying a function as the "bug." This work relies on the structure of the CFGs of both the application's functions and the code being searched for, which can significantly change with software version or obfuscation. IOVFI is resilient to such differences as long as the function's semantics remain the same. Unfortunately, we were also unable to obtain source code or detailed results for comparison.

DyCLINK [196] use dynamic analysis to compute a dependency graph between instructions executed during developer supplied unit tests. Code similarity is determined by computing an isomorphism between sub graphs, using edit distance between PageRank [197] vectors. DyCLINK targets Java applications so we cannot compare our prototype against it. DyCLINK considers methods as similar if they share *any* sufficiently similar behavior for a given input, an event much more prevalent in C binaries than Java binaries. Many dissimilar C functions behave similarly when handling errors (i.e., returning -1 on invalid input), while Java often favors raising different exceptions based on the error condition. We, therefore, believe that the common error handling technique in C would significantly affect DyCLINK's precision. IOVFI is able to distinguish between functions with similar functionality, because the decision tree, which encodes semantic similarity, is generated using *differences* in behavior.

Due to the diverse toolchains and architectures used and its closed source nature, binary analysis is particularly well suited to firmware. David et al. [195], created a static analysis

tool to find CVEs in firmwares, and discovered hundreds of vulnerabilities. Feng et al. [194], took inspiration from image search research to find bugs in Internet of Things devices by converting CFGs into numerical vectors for similarity analysis.

Neural network approaches Recently, neural networks have been used in binary analysis. Zuo et al. [17], trained a neural network to determine cross-architecture semantics of basic blocks. Liu et al. [23], employ a deep neural network to extract features from functions and the binary call graph. These features are then used to create a distance metric for determining binary similarity. Xu et al. [198] use a neural network to compute the embedding of a function's CFG to accelerate similarity computation. These approaches show promise in improving computer security by utilizing research from other research areas. However, it is unclear if their techniques remain accurate in the presence of different compilation environments.

5. SUMMARY

Software systems today are written as monoliths, meaning that instructions have full access to all addressable memory in the system. As a result of this monolithic design, hundreds of high severity CVEs get issued every year for the most crucial pieces of software. Compartmentalization is a well known security principle that is not applied in the majority of computer systems, but can limit the damage a bug in a system can cause. While implementing compartmentalization in a monolithic system is challenging, the security gains are large, especially in high privilege environments like kernels and hypervisors.

To address the lack of compartmentalization, this dissertation proposes the following systems: 1) HAKC for enforcing an arbitrary compartmentalization policy; 2) FlexC for discovering a compartmentalization policy based on type and call information, which can be gathered dynamically or statically; and 3) IOVFI for semantic function identification, which allows for exploring different compartmentalization strategies based on code similarity. These systems use external program state to achieve their goals of compartmentalization and semantic identification, and significantly advances the state-of-the-art in their respective areas.

HAKC makes use of memory tagging and pointer authentication to associate data ownership information with pointers, and enforces a compartmentalization policy by ensuring that all accessed data must be allowable as defined by the policy. HAKC automatically inserts the necessary checks and data ownership transfer operations in the generated code at compile time, negating the use of an extra layer of trust, such as a hypervisor. We compartmentalized the `ipv6.ko` and `nf_tables.ko` Linux kernel modules, and the overhead generated by HAKC is small (as little as 1.6%). In real-world browsing experiments, we measured no noticeable difference in user experience when other network effects are accounted for.

FlexC computes a Call-and-Type Graph for the whole kernel, and computes edge weights based on the static and dynamic information provided as input. The CTG is then used to derive a compartmentalization that suits the user's security or performance requirements. We evaluate the compartmentalizations based on performance overhead using the Linux Testing Project. Additionally, we evaluate the security gains by measuring the privilege

reduction achieved relative to the unmodified kernel, as well as a CVE study that measures whether an attack could be mitigated by a compartmentalization.

Finally, we propose IOVFI as a binary analysis framework that can be used for further exploration of compartmentalization strategies, in addition to the many other uses a semantic function identifier provides. IOVFI uses program state modifications, encoded as IOVecs, as the unique function fingerprint, as opposed to models computed from code properties. Unlike code property models, IOVecs can be sorted for quick query of unknown functions, and, using a small translation layer, can be used to identify functions in different architectures. IOVFI is significantly more accurate than the state-of-the-art static and dynamic semantic identifiers, especially when compilation environments of training and test binaries differ.

REFERENCES

[1] A. Carleton, J. Robert, M. Klein, and E. Harper, *Software engineering as a strategic advantage: A national roadmap for the future*, Carnegie Mellon University's Software Engineering Institute Blog, Nov. 2021. [Online]. [Online]. Available: http://insights.sei.cmu.edu/blog/software-engineering-as-a-strategic-advantage-a-national-roadmap-for-the-future/.

[2] C. Herley and P. C. Van Oorschot, "Sok: Science, security and the elusive goal of security as a scientific pursuit," in *2017 IEEE symposium on security and privacy (SP)*, IEEE, 2017, pp. 99–120.

[3] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[4] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Everything you want to know about pointer-based checking," in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[5] K. Cook, "Kernel address space layout randomization," *Linux Security Summit*, 2013.

[6] C. Cowan, C. Pu, D. Maier, *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.," in *USENIX security symposium*, San Antonio, TX, vol. 98, 1998, pp. 63–78.

[7] Google. "Oss-fuzz." (2018), [Online]. Available: https://github.com/google/oss-fuzz.

[8] J. Criswell, N. Dautenhahn, and V. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," 2014. DOI: 10.1109/SP.2014.26.

[9] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, 1974.

[10] R. M. Needham, "Protection systems and protection implementations," in *AFIPS '72 (Fall, part I)*, 1972.

[11] Z. Durumeric, F. Li, J. Kasten, *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14, Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488, ISBN: 9781450332132. DOI: 10.1145/2663716.2663755. [Online]. Available: https://doi.org/10.1145/2663716.2663755.

[12] N. Roessler, L. Atayde, I. Palmer, *et al.*, "μScope: A methodology for analyzing least-privilege compartmentalization in large software artifacts," in *24nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2021)*, USENIX Association, 2021.

[13] U. Bhatt, A. Xiang, S. Sharma, *et al.*, "Explainable machine learning in deployment," in *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, ser. FAT* '20, Barcelona, Spain: Association for Computing Machinery, 2020, pp. 648–657, ISBN: 9781450369367. DOI: 10.1145/3351095.3375624. [Online]. Available: https://doi.org/10.1145/3351095.3375624.

[14] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, 2016, pp. 24–35.

[15] R. Johnson and A. Stavrou, "Forced-path execution for android applications on x86 platforms," in *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, IEEE, 2013, pp. 188–197.

[16] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 829–844.

[17] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[18] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, Seattle, WA, USA: ACM, 2016, pp. 678–689, ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950350. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950350.

[19] U. Karg é n and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017, Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 342–352, ISBN: 978-1-5386-2684-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155608.

[20] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *ACM SIGPLAN Notices*, ACM, vol. 52, 2017, pp. 79–94.

[21] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: ACM, 2016, pp. 266–280, ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908126. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908126.

[22] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 709–724.

[23] B. Liu, W. Huo, C. Zhang, *et al.*, "α diff: Cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, Montpellier, France: ACM, 2018, pp. 667–678, ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238199. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238199.

[24] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, 2017, pp. 253–270, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ming.

[25] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2017, pp. 177–189.

[26] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, Leipzig, Germany: ACM, 2008, pp. 321–330, ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368132. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368132.

[27] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in cots binaries," in *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, IEEE, 2017, pp. 201–212.

[28] T. Reps, "Undecidability of context-sensitive data-dependence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 162–186, Jan. 2000, ISSN: 0164-0925. DOI: 10.1145/345099.345137. [Online]. Available: https://doi.org/10.1145/345099.345137.

[29] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2018, ISBN: 198508659X.

[30] H. Okhravi, "A cybersecurity moonshot," *IEEE Security & Privacy*, vol. 19, no. 3, pp. 8–16, 2021. DOI: 10.1109/MSEC.2021.3059438.

[31] NIST. "Cve-2016-4997 detail." (2021), [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016-4997.

[32] "The kernel address sanitizer (kasan)," 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html?highlight=kasan.

[33] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, 2016, pp. 179–194. DOI: 10.1109/EuroSP.2016.24.

[34] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "Aces: Automatic compartments for embedded systems," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 65–82.

[35] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Minimal kernel: An operating system architecture for TEE to resist board level physical attacks," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 105–120, ISBN: 978-1-939133-07-6. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/zhao.

[36] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "Xmp: Selective memory protection for kernel and user space," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 563–577. DOI: 10.1109/SP40000.2020.00041.

[37] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou, "Secpod: A framework for virtualization-based security systems," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA: USENIX Association, Jul. 2015, pp. 347–360, ISBN: 978-1-931971-225. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/wang-xiaoguang.

[38] L. Shi, Y. Wu, Y. Xia, *et al.*, "Deconstructing xen.," in *NDSS*, 2017.

[39] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as processes," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18, Carlsbad, CA, USA: Association for Computing Machinery, 2018, pp. 199–211, ISBN: 9781450360111. DOI: 10.1145/3267809.3267845. [Online]. Available: https://doi.org/10.1145/3267809.3267845.

[40] G. Klein, K. Elphinstone, G. Heiser, *et al.*, "Sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09, Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220, ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. [Online]. Available: https://doi.org/10.1145/1629575.1629596.

[41] D. Hildebrand, "An architectural overview of qnx," in *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, USA: USENIX Association, 1992, pp. 113–126, ISBN: 1880446421.

[42] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *Security Privacy, IEEE*, vol. 12, no. 2, pp. 16–26, Mar. 2014, ISSN: 1540-7993. DOI: 10.1109/MSP.2013.137.

[43] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "Kaslr: Break it, fix it, repeat," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20, Taipei, Taiwan: Association for Computing Machinery, 2020, pp. 481–493, ISBN: 9781450367509. DOI: 10.1145/3320269.3384747. [Online]. Available: https://doi.org/10.1145/3320269.3384747.

[44] L. ARM. "Pointer authentication on armv8.3." (2017), [Online]. Available: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-%20authentication-on-armv8-3.pdf.

[45] R. Avanzi, "The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 1, pp. 4–44, Mar. 2017. DOI: 10.13154/tosc.v2017.i1.4-44. [Online]. Available: https://tosc.iacr.org/index.php/ToSC/article/view/583.

[46] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "Pac it up: Towards pointer integrity using arm pointer authentication," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19, Santa Clara, CA, USA: USENIX Association, 2019, pp. 177–194, ISBN: 9781939133069.

[47] R. M. farkhani, M. Ahmadi, and L. Lu, "Ptauth: Temporal memory safety via robust points-to authentication," in *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C.: USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade.

[48] H. Liljestrand, Z. Gauhar, T. Nyman, J.-E. Ekberg, and N. Asokan, "Protecting the stack with paced canaries," in *Proceedings of the 4th Workshop on System Software for Trusted Execution*, ser. SysTEX '19, Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, ISBN: 9781450368889. DOI: 10.1145/3342559.3365336. [Online]. Available: https://doi.org/10.1145/3342559.3365336.

[49] R. Denis-Courmont, H. Liljestrand, C. Chinea, and J. .-. Ekberg, "Camouflage: Hardware-assisted cfi for the arm linux kernel," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218535.

[50] L. ARM. "Armv8.5-a memory tagging extension." (2019), [Online]. Available:

.

[51] C. Spensky, A. Machiry, N. Redini, *et al.*, "Conware: Automated modeling of hardware peripherals," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 95–109, ISBN: 9781450382878.

[52] C. Spensky, A. Machiry, N. Burow, *et al.*, "Glitching demystified: Analyzing control-flow-based glitching attacks and defenses," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 400–412. DOI: 10.1109/DSN48987.2021.00051.

[53] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[54] M. Lipp, M. Schwarz, D. Gruss, *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[55] Y. Kim, R. Daly, J. Kim, *et al.*, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210.

[56] J. M. McCune, A. Perrig, A. Seshadri, and L. van Doorn, "Turtles all the way down: Research challenges in user-based attestation," in *2nd USENIX Workshop on Hot Topics in Security (HotSec 07)*, Boston, MA: USENIX Association, Aug. 2007. [Online]. Available: https://www.usenix.org/conference/hotsec-07/turtles-all-way-down-research-challenges-user-based-attestation.

[57] B. Yee, D. Sehr, G. Dardyk, *et al.*, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 79–93. DOI: 10.1109/SP.2009.25.

[58] D. Schrammel, S. Weiser, S. Steinegger, *et al.*, "Donky: Domain keys – efficient in-process isolation for risc-v and x86," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 1677–1694, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel.

[59] M. Hedayati, S. Gravani, E. Johnson, *et al.*, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 489–504, ISBN: 978-1-939133-03-8. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/hedayati-hodor.

[60] V. Narayanan, A. Balasubramanian, C. Jacobsen, *et al.*, "Lxds: Towards isolation of kernel subsystems," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 269–284, ISBN: 978-1-939133-03-8. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/narayanan.

[61] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework?" *Queue*, vol. 11, no. 11, pp. 30–44, Dec. 2013, ISSN: 1542-7730. DOI: 10.1145/2557963.2566628. [Online]. Available: https://doi.org/10.1145/2557963.2566628.

[62] A. Madhavapeddy, R. Mortier, C. Rotsos, *et al.*, "Unikernels: Library operating systems for the cloud," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472, ISBN: 9781450318709. DOI: 10.1145/2451116.2451167. [Online]. Available: https://doi.org/10.1145/2451116.2451167.

[63] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04, Palo Alto, California: IEEE Computer Society, 2004, ISBN: 0-7695-2102-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673.

[64] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell, "Tetris is hard, even to approximate," in *Proceedings of the 9th Annual International Conference on Computing and Combinatorics*, ser. COCOON'03, Big Sky, MT, USA: Springer-Verlag, 2003, pp. 351–363, ISBN: 3540405348.

[65] NIST. "Cve-2017-9074 detail." (2021), [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-9074.

[66] NIST. "Cve-2019-14815 detail." (2021), [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-14815.

[67] J. A. Kroll, G. Stewart, and A. W. Appel, "Portable software fault isolation," in *2014 IEEE 27th Computer Security Foundations Symposium*, 2014, pp. 18–32. DOI: 10.1109/CSF.2014.10.

[68] Google. "Syzkaller." (2016), [Online]. Available: https://github.com/google/syzkaller.

[69] J. Gu, X. Wu, W. Li, *et al.*, "Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pp. 401–417, ISBN: 978-1-939133-14-4. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/gu.

[70] V. Narayanan, Y. Huang, G. Tan, T. Jaeger, and A. Burtsev, "Lightweight kernel isolation with virtualization and vm functions," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 157–171, ISBN: 9781450375542. DOI: 10.1145/3381052.3381328. [Online]. Available: https://doi.org/10.1145/3381052.3381328.

[71] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 191–206, ISBN: 9781450328357. DOI: 10.1145/2694344.2694386. [Online]. Available: https://doi.org/10.1145/2694344.2694386.

[72] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, "Enforcing least privilege memory views for multithreaded applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 393–405, ISBN: 9781450341394. DOI: 10.1145/2976749.2978327. [Online]. Available: https://doi.org/10.1145/2976749.2978327.

[73] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel, "Light-weight contexts: An OS abstraction for safety and performance," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 49–64, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton.

[74] S. Liu, D. Zeng, Y. Huang, *et al.*, "Program-mandering: Quantitative privilege separation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 1023–1040, ISBN: 9781450367479. DOI: 10.1145/3319535.3354218. [Online]. Available: https://doi.org/10.1145/3319535.3354218.

[75] S. Narayan, C. Disselkoen, T. Garfinkel, *et al.*, "Retrofitting fine grain isolation in the firefox renderer," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 699–716, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan.

[76] T. GARFINKEL, S. NARAYAN, C. DISSELKOEN, H. SHACHAM, and D. STEFAN, "The road to less trusted code," *USENIX PATRONS*, p. 15, 2020.

[77] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion, "Enclosure: Language-based restriction of untrusted libraries," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021, Virtual, USA: Association for Computing Machinery, 2021, pp. 255–267, ISBN: 9781450383172. DOI: 10.1145/3445814.3446728. [Online]. Available: https://doi.org/10.1145/3445814.3446728.

[78] M. Bauer and C. Rossow, "Cali: Compiler-assisted library isolation," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21, Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 550–564, ISBN: 9781450382878. DOI: 10.1145/3433210.3453111. [Online]. Available: https://doi.org/10.1145/3433210.3453111.

[79] A. Kurmus and R. Zippel, "A tale of two kernels: Towards ending kernel hardening wars with split kernel," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 1366–1377, ISBN: 9781450329576. DOI: 10.1145/2660267.2660331. [Online]. Available: https://doi.org/10.1145/2660267.2660331.

[80] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95, Copper Mountain, Colorado, USA: Association for Computing Machinery, 1995, pp. 251–266, ISBN: 0897917154. DOI: 10.1145/224056.224076. [Online]. Available: https://doi.org/10.1145/224056.224076.

[81] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 291–304, ISBN: 9781450302661. DOI: 10.1145/1950365.1950399. [Online]. Available: https://doi.org/10.1145/1950365.1950399.

[82] V. A. Sartakov, L. Vilanova, and P. Pietzuch, "Cubicleos: A library os with software componentisation for practical isolation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021, Virtual, USA: Association for Computing Machinery, 2021, pp. 546–558, ISBN: 9781450383172. DOI: 10.1145/3445814.3446731. [Online]. Available: https://doi.org/10.1145/3445814.3446731.

[83] A. Kivity, D. Laor, G. Costa, *et al.*, "Osv—optimizing the operating system for virtual machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA: USENIX Association, Jun. 2014, pp. 61–72, ISBN: 978-1-931971-10-2. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

[84] H. Lefeuvre, V.-A. Bădoiu, Ş. Teodorescu, *et al.*, "Flexos: Making os isolation flexible," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21, Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 79–87, ISBN: 9781450384384. DOI: 10.1145/3458336.3465292. [Online]. Available: https://doi.org/10.1145/3458336.3465292.

[85] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, *et al.*, "Unikraft: Fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 376–394, ISBN: 9781450383349. [Online]. Available: https://doi.org/10.1145/3447786.3456248.

[86] A. Madhavapeddy, T. Leonard, M. Skjegstad, *et al.*, "Jitsu: Just-in-time summoning of unikernels," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA: USENIX Association, May 2015, pp. 559–573, ISBN: 978-1-931971-218. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy.

[87] M. Compastié, R. Badonnel, O. Festor, R. He, and M. Kassi-Lahlou, "Unikernel-based approach for software-defined security in cloud infrastructures," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–7. DOI: 10.1109/NOMS.2018.8406155.

[88] V. Cozzolino, A. Y. Ding, and J. Ott, "Fades: Fine-grained edge offloading with unikernels," in *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, ser. HotConNet '17, Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 36–41, ISBN: 9781450350587. DOI: 10.1145/3094405.3094412. [Online]. Available: https://doi.org/10.1145/3094405.3094412.

[89] S. Lankes, J. Breitbart, and S. Pickartz, "Exploring rust for unikernel development," in *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, ser. PLOS'19, Huntsville, ON, Canada: Association for Computing Machinery, 2019, pp. 8–15, ISBN: 9781450370172. DOI: 10.1145/3365137.3365395. [Online]. Available: https://doi.org/10.1145/3365137.3365395.

[90] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 143–156, ISBN: 9781450375542. DOI: 10.1145/3381052.3381326. [Online]. Available: https://doi.org/10.1145/3381052.3381326.

[91] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong, "Theseus: An experiment in operating system structure and state management," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 1–19, ISBN: 978-1-939133-19-9. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/boos.

[92] V. Narayanan, T. Huang, D. Detweiler, *et al.*, "Redleaf: Isolation and communication in a safe operating system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 21–39, ISBN: 978-1-939133-19-9. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram.

[93] H. Okhravi, N. Burow, R. Skowyra, *et al.*, "One giant leap for computer security," *Security Privacy, IEEE*, 2020.

[94] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, "Keeping Safe Rust Safe with Galeed," in *Proceedings of IEEE Annual Computer Security Applications Conference (ACSAC'21)*, Dec. 2021.

[95] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238, ISBN: 978-1-939133-06-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[96] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "Libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 241–254, ISBN: 978-1-939133-03-8. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon.

[97] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX Association, Jul. 2021, pp. 805–820, ISBN: 978-1-939133-23-6. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/kotni.

[98] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on pku-based memory isolation systems," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 1409–1426, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/connor.

[99] Intel. "Intel® software guard extensions." (), [Online]. Available: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[100] C. Zhao, D. Saifuding, H. Tian, Y. Zhang, and C. Xing, "On the performance of intel sgx," in *2016 13th Web Information Systems and Applications Conference (WISA)*, 2016, pp. 184–187. DOI: 10.1109/WISA.2016.45.

[101] L. Mogosanu, A. Rane, and N. Dautenhahn, "Microstache: A lightweight execution context for in-process safe region isolation," in *Research in Attacks, Intrusions, and Defenses*, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds., Cham: Springer International Publishing, 2018, pp. 359–379, ISBN: 978-3-030-00470-5.

[102] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-process memory isolation extension," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 83–97, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/frassetto.

[103] C. Song, H. Moon, M. Alam, *et al.*, "Hdfi: Hardware-assisted data-flow isolation," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 1–17. DOI: 10.1109/SP.2016.9.

[104] S. Xu, W. Huang, and D. Lie, "In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021, Virtual, USA: Association for Computing Machinery, 2021, pp. 224–240, ISBN: 9781450383172. DOI: 10.1145/3445814.3446761. [Online]. Available: https://doi.org/10.1145/3445814.3446761.

[105] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the c programming language," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 103–114, Mar. 2008, ISSN: 0163-5980. DOI: 10.1145/1353535.1346295. [Online]. Available: https://doi-org.libproxy.mit.edu/10.1145/1353535.1346295.

[106] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, Belgrade, Serbia: Association for Computing Machinery, 2017, pp. 437–452, ISBN: 9781450349383. DOI: 10.1145/3064176.3064217. [Online]. Available: https://doi.org/10.1145/3064176.3064217.

[107] N. Roessler and A. DeHon, "Protecting the stack with metadata policies and tagged hardware," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 478–495. DOI: 10.1109/SP.2018.00066.

[108] E. Witchel, J. Rhee, and K. Asanović, "Mondrix: Memory isolation for linux using mondriaan memory protection," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05, Brighton, United Kingdom: Association for Computing Machinery, 2005, pp. 31–44, ISBN: 1595930795. DOI: 10.1145/1095810.1095814. [Online]. Available: https://doi.org/10.1145/1095810.1095814.

[109] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X, San Jose, California: Association for Computing Machinery, 2002, pp. 304–316, ISBN: 1581135742. DOI: 10.1145/605397.605429. [Online]. Available: https://doi.org/10.1145/605397.605429.

[110] J. Woodruff, R. N. Watson, D. Chisnall, *et al.*, "The cheri capability model: Revisiting risc in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, IEEE, 2014, pp. 457–468.

[111] ARM. "Trustzone." (), [Online]. Available: https://developer.arm.com/ip-products/security-ip/trustzone.

[112] D. Chu, Y. Wang, L. Lei, Y. Li, J. Jing, and K. Sun, "Ocram-assisted sensitive data protection on arm-based platform," in *Computer Security – ESORICS 2019*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds., Cham: Springer International Publishing, 2019, pp. 412–438, ISBN: 978-3-030-29962-0.

[113] S. Wan, M. Sun, K. Sun, N. Zhang, and X. He, "Rustee: Developing memory-safe arm trustzone applications," in *Annual Computer Security Applications Conference*, ser. ACSAC '20, Austin, USA: Association for Computing Machinery, 2020, pp. 442–453, ISBN: 9781450388580. DOI: 10.1145/3427228.3427262. [Online]. Available: https://doi-org.libproxy.mit.edu/10.1145/3427228.3427262.

[114] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves," in *Network and Distributed Systems Security (NDSS) Symposium*, Feb. 2019. DOI: 10.14722/ndss.2019.23448.

[115] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, "Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v," in *In Network and Distributed System Security Symposium (NDSS)*, 2019. DOI: 10.14722/ndss.2019.23068.

[116] C. H. Kim, T. Kim, H. Choi, *et al.*, "Securing real-time microcontroller systems through customized memory view switching," in *Network and Distributed Systems Security (NDSS) Symposium*, Feb. 2018, ISBN: 1-1891562-49-5. DOI: 10.14722/ndss.2018.23107.

[117] L. Davi, M. Hanreich, D. Paul, *et al.*, "Hafix: Hardware-assisted flow integrity extension," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6. DOI: 10.1145/2744769.2744847.

[118] A. A. Clements, N. S. Almakhdhub, K. S. Saab, *et al.*, "Protecting bare-metal embedded systems with privilege overlays," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 289–303. DOI: 10.1109/SP.2017.37.

[119] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "Pacstack: An authenticated call stack," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand.

[120] Clang. "Hardware-assisted addresssanitizer design documentation." (), [Online]. Available: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[121] R. Nikolaev and G. Back, "Virtuos: An operating system with kernel virtualization," in *Proceedings of the Twenty-Fourth ACMSymposium on Operating Systems Principles*, ser. SOSP '13, Farminton,Pennsylvania: Association forComputing Machinery, 2013, pp. 116–132, ISBN: 9781450323888. DOI: 10.1145/2517349.2522719. [Online]. Available: https://doi.org/10.1145/2517349.2522719.

[122] K. Fraser, S. H, R. Neugebauer, I. Pratt, and M. Williamson, "Safe hardware access with the xen virtual machine monitor," in *In Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[123] X. Xiong, D. Tian, and P. Liu, "Practical protection of kernel integrity for commodity os from untrusted extensions," in *NDSS*, 2011.

[124] T. D. Ngoc, B. Teabe, A. Tchana, G. Muller, and D. Hagimont, "Mitigating vulnerability windows with hypervisor transplant," in *Proceedings of the Sixteenth European Conference on Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 162–177, ISBN: 9781450383349. [Online]. Available: https://doi.org/10.1145/3447786.3456235.

[125] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "A secure and formally verified linux kvm hypervisor," 2021.

[126] K. Huang, Y. Huang, M. Payer, *et al.*, "The taming of the stack: Isolating stack data from memory errors," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2022, p. 17.

[127] G. Chen, H. Jin, D. Zou, *et al.*, "Safestack: Automatically patching stack-based buffer overflow vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 6, pp. 368–379, 2013. DOI: 10.1109/TDSC.2013.25.

[128] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 985–999. DOI: 10.1109/SP.2019.00076.

[129] N. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "Urai: Return address integrity for embedded systems," in *In Network and Distributed System Security Symposium (NDSS)*, 2020. DOI: 10.14722/ndss.2020.24016.

[130] N. Roessler and A. DeHon, "Protecting the stack with metadata policies and tagged hardware," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 478–495. DOI: 10.1109/SP.2018.00066.

[131] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "{Pacstack}: An authenticated call stack," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 357–374.

[132] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 1867–1881, ISBN: 9781450367479. DOI: 10.1145/3319535.3354244. [Online]. Available: https://doi.org/10.1145/3319535.3354244.

[133] S. Pfleeger and R. Cunningham, "Why measuring security is hard," *IEEE Security Privacy*, vol. 8, no. 4, pp. 46–54, 2010. DOI: 10.1109/MSP.2010.60.

[134] P. Biswas, N. Burow, and M. Payer, "Code specialization through dynamic feature observation," in *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 257–268, ISBN: 9781450381437. DOI: 10.1145/3422337.3447844. [Online]. Available: https://doi.org/10.1145/3422337.3447844.

[135] M. Zhang and R. Sekar, "Control flow integrity for {cots} binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[136] H. Kellerer, U. Pferschy, and D. Pisinger, "Multiple knapsack problems," in *Knapsack Problems*, Springer, 2004, pp. 285–316.

[137] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, pp. 1157–1177, Dec. 2017, ISSN: 0098-5589. DOI: 10.1109/TSE.2017.2655046. [Online]. Available: https://doi.org/10.1109/TSE.2017.2655046.

[138] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09, Chicago, IL, USA: ACM, 2009, pp. 117–128, ISBN: 978-1-60558-338-9. DOI: 10.1145/1572272.1572287. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572287.

[139] T. Rid and B. Buchanan, "Attributing cyber attacks," *Journal of Strategic Studies*, vol. 38, no. 1-2, pp. 4–37, 2015.

[140] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C.: USENIX, 2013, pp. 81–96, ISBN: 978-1-931971-03-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/jang.

[141] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering.," in *NDSS*, Citeseer, vol. 9, 2009, pp. 8–11.

[142] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 143–157, ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.17. [Online]. Available: https://doi.org/10.1109/SP.2008.17.

[143] R. Duan, A. Bijlani, Y. Ji, *et al.*, "Automating patching of vulnerable open-source software versions in application binaries," in *NDSS*, 2019.

[144] D. Mantz, J. Classen, M. Schulz, and M. Hollick, "Internalblue-bluetooth binary patching and experimentation framework," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019, pp. 79–90.

[145] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1253–1270.

[146] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 151–163, ISBN: 9781450376136. DOI: 10.1145/3385412.3385972. [Online]. Available: https://doi.org/10.1145/3385412.3385972.

[147] S. Almanee, A. Unal, M. Payer, and J. Garcia, "Too quiet in the library: An empirical study of security updates in android apps' native code," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 1347–1359. DOI: 10.1109/ICSE43902.2021.00122. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE43902.2021.00122.

[148] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002, ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1019480. [Online]. Available: https://doi.org/10.1109/TSE.2002.1019480.

[149] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code.."

[150] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105, ISBN: 0-7695-2828-7. DOI: 10.1109/ICSE.2007.30. [Online]. Available: https://doi.org/10.1109/ICSE.2007.30.

[151] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09, Chicago, IL, USA: ACM, 2009, pp. 81–92, ISBN: 978-1-60558-338-9. DOI: 10.1145/1572272.1572283. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572283.

[152] Y. Shoshitaishvili, R. Wang, C. Salls, *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 138–157.

[153] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, 2014, pp. 303–317, ISBN: 978-1-931971-15-7. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele.

[154] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017, Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 319–330, ISBN: 978-1-5386-2684-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155606.

[155] zynamics. "Bindiff." (2020), [Online]. Available: https://www.zynamics.com/bindiff.html.

[156] S. HexRays. "Interactive disassembler." (2019), [Online]. Available: https://www.hex-rays.com/index.shtml.

[157] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489. DOI: 10.1109/SP.2019.00003.

[158] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, ser. WORM '07, Alexandria, Virginia, USA: ACM, 2007, pp. 46–53, ISBN: 978-1-59593-886-2. DOI: 10.1145/1314389.1314399. [Online]. Available: http://doi.acm.org/10.1145/1314389.1314399.

[159] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, IEEE, 2006, pp. 289–300.

[160] radare. "Radare." (2019), [Online]. Available: https://www.radare.org/r/.

[161] musl libc. "Musl libc." (2019), [Online]. Available: https://www.musl-libc.org/.

[162] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08, Alexandria, Virginia, USA: ACM, 2008, pp. 51–62, ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455779. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455779.

[163] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1083–1094.

[164] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, 2008, pp. 209–224.

[165] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" In *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 181–198, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.

[166] M. Zalewski. "American fuzzy lop." (2015), [Online]. Available: http://lcamtuf.coredump.cx/afl/.

[167] Google. "Honggfuzz." (2018), [Online]. Available: https://github.com/google/honggfuzz.

[168] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "Kafl: Hardware-assisted feedback fuzzing for OS kernels," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, 2017, pp. 167–182, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

[169] W. Xu, H. Moon, S. Kashyap, P. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *2019 2019 IEEE Symposium on Security and Privacy (SP)*, vol. 00, 2017, pp. 577–593. DOI: 10.1109/SP.2019.00035. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2019.00035.

[170] G. Grieco, M. n. Ceresa, and P. Buiras, "Quickfuzz: An automatic random fuzzer for common file formats," in *Proceedings of the 9th International Symposium on Haskell*, ser. Haskell 2016, Nara, Japan: ACM, 2016, pp. 13–20, ISBN: 978-1-4503-4434-0. DOI: 10.1145/2976002.2976017. [Online]. Available: http://doi.acm.org/10.1145/2976002.2976017.

[171] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, Tucson, AZ, USA: ACM, 2008, pp. 206–215, ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375607. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375607.

[172] J. Corina, A. Machiry, C. Salls, *et al.*, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: ACM, 2017, pp. 2123–2138, ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134069. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134069.

[173] D. Song, F. Hetzelt, D. Das, *et al.*, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *2019 Network and Distributed Systems Security Symposium (NDSS)*, Internet Society, 2019, pp. 1–15.

[174] 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.

[175] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 711–725.

[176] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/.

[177] LLVM. "Libfuzzer." (2019), [Online]. Available: https://llvm.org/docs/LibFuzzer.html.

[178] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015, Pittsburgh, PA, USA: ACM, 2015, pp. 386–399, ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814319. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814319.

[179] P. Chen, J. Liu, and H. Chen, "Matryoshka: Fuzzing deeply nested branches," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 499–513, ISBN: 9781450367479. DOI: 10.1145/3319535.3363225. [Online]. Available: https://doi.org/10.1145/3319535.3363225.

[180] R. Milewicz, R. Vanka, J. Tuck, D. Quinlan, and P. Pirkelbauer, "Runtime checking c programs," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ACM, 2015, pp. 2107–2114.

[181] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007, ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. [Online]. Available: https://doi.org/10.1145/1273442.1250746.

[182] R. M. Stallman and G. DeveloperCommunity, *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009, ISBN: 144141276X, 9781441412768.

[183] N. S. Agency. "Ghidra." (2019), [Online]. Available: https://www.nsa.gov/resources/everyone/ghidra/.

[184] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACM, 2016, pp. 189–200.

[185] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the semantics of obfuscated code," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 643–659, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko.

[186] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator- LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15 , Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed., IEEE, 2015, pp. 3–9. DOI: 10.1109/SPRO.2015.10.

[187] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014, pp. 281–293.

[188] R. Wang, Y. Shoshitaishvili, A. Bianchi, *et al.*, "Ramblr: Making reassembly great again," in *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17)*, 2017.

[189] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries.," in *NDSS*, Citeseer, 2015.

[190] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, Montpellier, France: ACM, 2018, pp. 896–899, ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3240480. [Online]. Available: http://doi.acm.org/10.1145/3238147.3240480.

[191] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "Discovre: Efficient cross-architecture identification of bugs in binary code," Feb. 2016. DOI: 10.14722/ndss.2016.23185.

[192] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun, "Eclone: Detect semantic clones in ethereum via symbolic transaction sketch," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: ACM, 2018, pp. 900–903, ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3264596. [Online]. Available: http://doi.acm.org/10.1145/3236024.3264596.

[193] A. Saebjornsen, *Detecting fine-grained similarity in binaries*. University of California, Davis, 2014.

[194] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 480–491, ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978370. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978370.

[195] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2018, pp. 392–404.

[196] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, "Code relatives: Detecting similarly behaving software," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, Seattle, WA, USA: ACM, 2016, pp. 702–714, ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950321. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950321.

[197] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Stanford InfoLab, Technical Report 1999-66, Nov. 1999, Previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/.

[198] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 363–376.