Practical Exploit Mitigation Design Against Code Re-Use and System Call Abuse Attacks

Christopher Stanislaw Jelesnianski

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Changwoo Min, Chair
Yeongjin Jang
Wenjie Xiong
Danfeng (Daphne) Yao
Haibo Zeng

December 2, 2022
Blacksburg, Virginia

Keywords: Exploit Mitigation, Practical Design, System Software, Randomization, Runtime Defense, Return-Oriented Programming, Code Re-use Attacks, System Calls, System Call Specialization

Practical Exploit Mitigation Design Against Code Re-Use and System Call Abuse Attacks

Christopher Stanislaw Jelesnianski

(ABSTRACT)

Practically is an important, yet overlooked quality in exploit mitigation design. While many defense techniques have been proposed, few have been adopted by the general public and deployed in production, specifically for their lack of practicality. With the limited amount of defenses deployed in mainstream computing, mediocre security is common. This has serious consequences, as large scale cyber-attacks are now a common occurance in society. Practical design is important in order to promote mainstream adoption and minimize adverse side-effects such as performance degradation or memory monopolization. To be practical, exploit mitigations should be performant, robust, scalable, and guarantee strong levels of security. However, balancing all four of these equally important goals is hard to achieve in practice.

Practical design must successfully navigate several difficult challenges. Modern attacks have become capable of leveraging many types of code components; correspondingly, defenses have extended their capabilities to protect more aspects of code. To sufficiently protect code, defense techniques have unintentionally become increasingly complex. Supporting such fine-grained defenses brings inherent disadvantages such as significant hardware resource utilization that could be otherwise used for useful work. Complexity has made performance, security, and scalability all competing ideals in practical system design. Other defenses have implemented mechanisms with negligible performance impact at the risk of weaker security guarantees. A practical defense would not sacrifice any design aspect to maintain other desirable properties.

Rather than continuing this status quo, this dissertation formalizes the challenges in modern exploit mitigation design and presents two *practical* design solutions. Both defense systems uphold goals of achieving *realistic performance*, providing *strong security guarantees*, *being robust* for modern application code bases, and being *able to scale* across the system at large.

This dissertation first goes over the core motivation for these contributions. Attackers accept that rudimentary defenses such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) will be deployed. Attacks have therefore evolved to diversify their angles of attack and are now capable of leveraging a multitude of different code components. Accordingly, the security community has uncovered these threats and maintained progress in providing possible resolutions with new exploit mitigation designs. This dissertation presents a survey from the perspective of both attacker and defender to provide an overview of this current security landscape. Specifically, to further support reader

understanding of this dissertation, this dissertation defines an informal taxonomy of exploit mitigation strategies, it explains prominent attack vectors that are faced by security experts today, and it identifies and defines code components that are generally abused by code re-use.

This dissertation then presents its first practical exploit mitigation design, MARDU. MARDU is a novel re-randomization approach that utilizes on-demand randomization and the concept of code trampolines to support sharing of code transparently system-wide. *To the best of my knowledge, MARDU is the first presented re-randomization technique capable of runtime code sharing for re-randomized code system-wide.* Moreover, MARDU is one of the very few re-randomization mechanisms *capable of performing seamless live thread migration to newly randomized code without pausing application execution.* This disseration describes the full design, implementation, and evaluation of MARDU to demonstrate its merits and show that careful design can uphold all practical design goals. For instance, scalability is a major challenge for randomization strategies, especially because traditional OS design expects code to be placed in known locations so that it can be reached by multiple processes, while randomization is purposefully trying the achieve the opposite, being completely unpredictable. This clash in expectations between system and defense design breaks a few very important assumptions for an application's runtime environment. This forces most randomization mechanisms to abandon the hope of upholding memory deduplication. MARDU resolves this challenge by applying trampolines to securely reach functions protected under secure memory. Even with this new calling convention in place, MARDU shows re-randomization degradation can be significantly reduced without sacrificing randomization entropy. Moreover, MARDU shows it is capable of defeating prominent code re-use variants with this practical design.

Finally, this dissertation presents its second practical exploit mitigation solution, BASTION. BASTION is a *fine-grained system call filtering mechanism* aimed at significantly strengthening the security surrounding system calls. Like MARDU, BASTION upholds the principles of this dissertation and was implemented with practicality in mind. BASTION's design is based on empirical observation of what a legitimate system call invocation consists of. BASTION introduces *System Call Integrity* to enforce the correct and intended use of system calls within a program. In order to enforce this novel security policy, BASTION proposes three new specialized contexts for the effective enforcement of legitimate system call usage. Namely, these contexts enforce that: system calls are only invoked with the correct calling convention, system calls are reached through legitimate control-flow paths, and all system call arguments are free from attacker corruption. By enforcing System Call Integrity with the previously mentioned contexts, this dissertation adds further evidence that context-*sensitive* defense strategies are superior to context-*insensitive* ones. BASTION is able to prevent over 32 real-world and synthesized exploits in its security evaluation and incurs negligible performance overhead (0.60%-2.01%). BASTION demonstrates that narrow and specialized exploit mitigation designs can be effective in more than one front, to the point that BASTION not only prevents code re-use, but is capable of defending against any attack class that requires the utilization of system calls.

Practical Exploit Mitigation Design Against Code Re-Use and System Call Abuse Attacks

Christopher Stanislaw Jelesnianski

(GENERAL AUDIENCE ABSTRACT)

Practicality is an important quality in security mechanisms for modern day computer systems. While many defense techniques have been proposed, few have become adopted by the general public. To elaborate, most defense systems are not immediately adopted because of the significant tradeoffs they put upon a user. Common tradeoffs include adverse side-effects such as slowing down user applications or imposing significant memory usage. Limited defense deployment and weak security has serious consequences, as large scale cyber-attacks are now a common occurance. Therefore, practical and strong defense design is important to promote integration into the next generation of computer hardware and software. By sustaining practical design, the needed jump between a proof-of-concept and implementing it on commodity computer chips is substantially smaller. A practical defense should foremost guarantee strong levels of security and should not slow down a user's applications. Ideally, a practical defense is implemented to the point it seems invisible to the user and they don't even notice it. However, balancing the goals in practicality is hard to achieve in practice.

This dissertation first reviews the current security landscape - specifically two important attack strategies are examined. First, code re-use attacks, are exactly what they sound like; code re-use essentially reuse various bits and pieces of program code to create an attack. Second, system call abuse. System calls are essential functions that ordinarily allow a user program to talk with a computer's operating system; they enable operations such as a program asking for more memory or reading & writing files. When system calls are maliciously abused, they can cause a computer to use up all its free memory or even launch an attacker-written program. This dissertation goes over how these attacks work and correspondingly explains popular defense strategies that have been proposed by the security community so far.

This dissertation then presents two defense system solutions that demonstrate how a practical defense system could be made. To that end, the full design, implementation, and evaluation of each defense systems is presented. This dissertation mostly leverages compiler techniques to achieve its goal. A compiler is an essential developer tool that converts human written code into a computer program. However, the compiler can also be used to apply additional optimizations and security hardening techniques to make a program more secure.

This dissertation presents MARDU, a runtime randomization defense. MARDU protects programs by randomizing the location of code chunks throughout execution so that attackers cannot find the code pieces they need to create an attack. Notably, MARDU is the first

randomization defense that is able to be seamlessly deployed system-wide and is backwards compatible with programs not outfitted with MARDU. This dissertation also presents BASTION, a defense system that strictly focuses on protection of system calls in a program. As mentioned earlier, system calls are security critical functions that allow a program to talk the computers operating system. BASTION protects the entire computer by ensuring that every time a system call is called by a user program, it was really requested by the program and not maliciously by an attacker. BASTION verifies this request is honest by confirming that the current program state meets a certain set of criteria.

Dedication

To my parents Stanisław and Elzbieta Jeleśniański, and my sister Joanna.

Acknowledgments

This dissertation would not have been possible if not for all the wonderful individuals that helped me along the winding way, full of twists and turns, and surprises that come with the territory of completing a PhD. It has definitely been an unforgetable experience.

First, I would like to sincerely thank my advisor, Professor Changwoo Min, for so much. Foremost for seeing the potential in me as a pupil in computer engineering and accepting me as a PhD candidate into the COSMOSS Lab as I transitioned from my Master's thesis. I sincerely appreciate all the time you spent by my side, especially at the beginning, mentoring me to not just be an ordinary computer engineer and researcher, but to always strive to be an exceptional one. I admire your patience, dedication, leadership, and kindness - your words of encouragement throughout this journey continuously supported my growth. You have shown me what it means to be truly dedicated to one's craft. I cannot thank you enough for all your support, help, and guidance you have given me during my time at Virginia Tech, and for all of time we spent together brainstorming at the whiteboard, finalizing paper revisions, and working through setbacks that arise in research - it was truly exciting to work with you.

In addition to Professor Min, I would like to express my gratitude to my committee members for serving on my committee: Professor Yeongjin Jang, Professor Wenjie Xiong, Professor Danfeng (Daphne) Yao, and Professor Haibo Zeng. Professor Yeongjin Jang, I am grateful for all you taught me about ROP and the security domain. Even though we mostly met across the internet and different time zones, your shown passion and spirit always always made it as if we were meeting in person.

Next I would like to thank my very first boss, Thomas McClaren Sr. You shared your passion of never-ending learning - as the ocean never rests, so too does the pace of innovation in technology. You introduced me to proper documentation and organization, to RTS (Read The Screen) to fix things, and celebrated even the smallest of victories in optimizations (F5) that aided our work. You fostered my love for computers and encouraged me to make my hobby of tinkering with computers more than just a hobby, rather I have now made it an entire career.

Next, I would like to thank all of my friends, colleagues, and labmates that I've had the pleasure of spending time with during my time at Virginia Tech. To Antonio Barbalace, the super amazing Italian Linux kernel and compiler wizard who could fix anything, thank you for being you. Thank you for all the mentoring you gave me, especially for my Master's Degree, and all the life advise we talked about thereafter. Your energy and enthusiasm (even at 2AM) was a huge boost to keep morale high and to never give up. To the SSRG compiler specialistics, Rob Lyerly, Ho-Ren (Jack) Chuang, Anthony Carno, and Ian Roessle. I will

not forget our late night debugging, research discussions, lengthy benchmarking sessions, and board game nights. To my COSMOSS lab friends Ajit Mathew, Madhava Krishnan, Jaeho Kim, Xinwei (Mason) Fu, Jinwoo Yom, and Mohannad Ismail. I am proud of how the COSMOSS lab has grown with all of your achievements. To all the graduates students I would regularly hang out with from the Virginia Tech Wireless Lab next door: Chris O'Lone, Will Howard, Charlie, Don, and Xavier. All of your camaraderie helped us all get through tough times and made celebrations all the more fun, grad school would not have been the same without you.

Lastly and most importantly, I could not have done this without my family, my mom Elzbieta, my dad Stanisław, and my sister Joanna. You would always check up on me and make sure everything was alright. You all helped me more than anyone, always rooting for me, always telling me to not get discouraged and persevere when things didn't go as planned, and always sending me home-made treats to have a taste of home. I could not have gotten through this chapter of my life without your constant and loving support. I love you!

Contents

List of Figures

List of Tables

List of Abbreviations

ASLR	Address Space Layout Randomization
BPF	Berkley Packet Filter
CET	Control-flow Enforcement Technology
CFG	Control-Flow Graph
CFI	Control-Flow Integrity
COOP	Counterfeit Object-Oriented Programming
COTS	Commercial Off the Shelf
CPI	Code Pointer Integrity
DEP	Data Execution Prevention
DFI	Data-Flow Integrity
EC	Equivalence Class
ELF	Executable and Linkable Format
FTP	File Transfer Protocol
`glibc`	GNU C Library
GOT	Global Offset Table
ICT	Indirect Control-flow Transfer
ISA	Instruction Set Architecture
I/O	Input/Output
JIT	Just-In-Time
LoC	Lines of Code
MPK	Memory Protection Keys
OS	Operating System
PAC	Pointer Authentication Code
PC	Program Counter
PLT	Procedural Linkage Table
PT	Processor Trace
ROP	Return-Oriented Programming
R$\oplus$X	Read exclusive or Execute
seccomp	Secure Computing
seccomp BPF	Secure Computing with filters
syscall	System Call
TOCTOU	Time-of-check-Time-of-use
W$\oplus$R	Write exclusive or Read
XoM	eXecute-Only Memory
X86	The X86 Instruction Set Architecture

Chapter 1

Introduction

1.1 Motivation

While many modern defense techniques have been recently proposed, very few have been actually adopted in mainstream computing or deployed in production. For this reason, attacks continue to gain ground on real-world applications whenever a new vulnerability is found. When C and C++ became defacto programming languages, many production applications were implemented in these new paradigms that are inherently not memory safe. At the same time, it was not realized that this was a major liability until early manifestations of memory safety vulnerabilities presented themselves, in the form of buffer overflows, format string vulnerabilities, and integer conversion vulnerabilities. These early memory safety bugs allowed malicious actors to perform operations such as overwriting data, corrupting and pivoting execution state (*e.g.*, the stack), and leaking sensitive information. To illustrate, stack smashing [7], abused the inherent nature of the x86 calling convention with the presence of a memory safety bug to directly overwrite the stack with attacker imposed values and enable control-flow to then be pivoted to follow the attackers will, while code injection [10, 113] modifies heap metadata in order to to change program execution. These early attack vectors reigned free until proper solutions (*e.g.*, Data Execution Prevention (DEP) [148], Address Space Layout Randomization (ASLR) [210]) were introduced and became widespread. Since then, attack vectors have increasingly become more versatile in their attack capability in order to circumvent these standard defense practices and evolved from direct memory disclosure [26, 194], to indirect memory disclosure [25, 185], to creating attack chains on the fly during runtime [201, 222].

To address this versatility among various attack vectors, exploit mitigation techniques have kept attacks at bay by extending their attack surface protection and increasingly protecting more application code. Adequate security is especially challenging to accomplish in a practical manner when so many code components are deemed sensitive and are required to be protected. In order to sustain sufficient protection of applications, defense techniques have gravitated towards more complex designs including re-randomization [22, 41, 45, 119, 123, 124, 141, 205, 224, 228, 232], data integrity [103, 126, 202, 203], memory safety [20, 43, 54, 59, 65, 152, 198, 199, 229, 239], and control-flow integrity [1, 77, 86, 87, 95, 117, 118, 139, 163, 219, 220]. Nonetheless, some straightforward approaches like information hiding [13, 91, 169, 225], debloating [3, 4, 8, 18, 78, 90, 125, 171, 175, 176, 180, 181, 183, 196, 237, 240], and system call

filtering [29, 34, 57, 79, 80, 104, 115, 150, 168] have been discovered along the way. Regardless if an exploit mitigation strategy is complex or simple, many approaches omit key properties of practicality. That being said, it is challenging to harmoniously integrate practicality into exploit mitigation design by skillfully balancing performance, strong security, scalability, and robustness, all at the same time. Strong exploit mitigations that have broad coverage such as CFI [1] or DFI [38] often have proportionally high performance overhead and require significant hardware resources to operate. By contrast, defenses that exhibit negligible or zero performance overhead, such as Address Space Layout Randomization (ASLR) [210] and system call filtering (*e.g.*, seccomp) [111, 115] are stuck with tradeoffs such as providing weaker security guarantees or being fragile to unexpected user input.

1.1.1 Prevalent Real-world Attack Scenarios

The intricate relationship between attack development and counter-active exploit mitigation design has always been a race. A race for the attacker to uncover the next unguarded code component, and correspondingly, a race for the defender to produce an adequate exploit mitigation mechanism that is not only successful in defending against the latest threat but practical enough to be deployed amongst computer systems worldwide. As mentioned earlier, the introduction of DEP largely eliminated code injection attacks, making code re-use attack variants the current predominately used attack methodology. Figure 1.1 shows the high level view of a conventional code re-use attack procedure. First, attacks assume there is an inherent vulnerability in a targt application. In most cases, these vulnerabilities originate from design flaws, human error, unintentional bugs in the form of memory safety vulnerabilities, unvalidated/unexpected user input, or absent error checking. These security vulnerabilities establish an opening or weak point in a target applications. Notably, these vulnerability can enable memory disclosure of the entire code layout, allowing an attacker to locate code components they require and chain them together to form a viable attack. As explained by Snow *et al.* [201], a single memory disclosure, or even an indirect memory disclosure (*e.g.*, Blind ROP [25]) can allow an attacker to acertain all necessary information. Once an attacker knows an applications specific memory layout, they can manipulate each code component at will, via corruption of code pointers and variable values. Finally, code re-use attacks leverage a system call to finalize an attack chain and achieve their intentions. Malicious attacker intentions can include operations such as privilege escalation by changing permission flags (*e.g.*, `setuid()`), making unintended memory regions executable (*e.g.*, `mprotect()`), or launching a new shell for arbitrary code execution (*e.g.*, `execve()`).

1.1.2 Modern Defenses

Instead of being spread thin attempting to protect everything, modern exploit mitigation designs have specialized into a few distinct archetypes. Specifically, these archetypes origi-

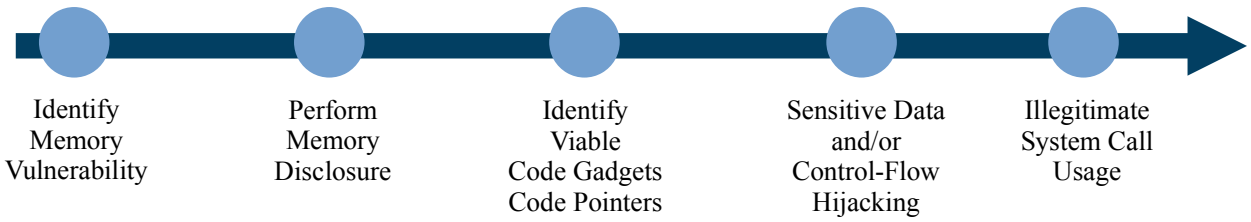


Figure 1.1: Conventional attack procedure to accomplish a code re-use attack

Attack Step	Presence of Sensitive Code Components	Identifying Memory Vulnerability	Perform Memory Disclosure	Identify Viable Code Components	Sensitive Data & Control-Flow Hijacking	Illegitimate System Call Usage
Corresponding Exploit Mitigation Archetype	Debloating	Memory Safety	Information Hiding	Re-Randomization	Data Integrity, Control Flow Integrity	System Call Filtering
Exploit Mitigation Examples	Piece-wise [181] RAZOR [180] CHISEL [90] Nibbler [3] TRIMMER [4] PacJam [171]	HardBound [59] SoftBound [152] BOGO [239] ViK [43] Oscar [54] CHERI [229]	Oxymoron [13] Readactor [46] HideM [81] NEAR [231] HeisenByte [207] kR X [174]	**MARDU** [107, 109] Shuffler [232] CodeArmor [41] ReRanz [224] TASR [22] RuntimeASLR [141]	DFI [38] CPI [126] VIP [103] CFI [1] μCFI [95] OS-CFI [118]	**BASTION** Saffire [150] `sysfilter` [57] Temporal [80] ABHAYA [168] Shredder [149]

Table 1.1: Table showing corresponding exploit mitigation design archetypes as well as listing notable works that implement each design archetype for each step performed by code re-use.

nate from the current understanding of what steps are necessary to complete a code re-use attack (recall Figure 1.1). Each distinct archetype blocks code re-use at a specific step and prevents it from proceding further. To illustrate this connection between code re-use and the defense archetypes designed to block them, we present Table 1.1. This table lists the modern exploit mitigation design archetype that is employed to block a particular code re-use step, as well as noting notable defenses that implement that particular archetype design. Each major exploit mitgitation archetype has its own respective advantages and disadvantages. For example, some exploit mitigation archetypes are complete and deterministic, however they often incur significant performance degradation. On the other hand, others have negligible performance impact but provide incomplete coverage and are able to be bypassed by more complex attack strategies. In reality, very few proposed defense techniques are able to excel in all aspects of practicality: performance, security, scalability, and robustness.

1.1.3 Challenges in Practical Exploit Mitigation Design

Regardless of the major archetype an exploit mitigation framework employs, each strategy experiences different challenges in practicality to various degrees when attempting to be implemented. Here, we explain the core challenges encountered when seeking to create new exploit mitigation designs that are practical.

- **Performance Tradeoffs** Tracing and updating various runtime information is an inherent part of many defense strategies. This is especially relevant for fine-grained

techniques that must keep track of many moving pieces throughout an application's runtime, either for validation purposes (*e.g.*, data integrity) [38, 126] or to maintain program semantics due to the constant churn of code and memory (*e.g.*, randomization) [22, 141, 232]. Ironically, in the case of randomization [74, 108, 224], exploit mitigation techniques must keep track of code chunks in the same way an attacker needs to track the same code chunks in an attempt to overcome this defense strategy. These factors often cause an exploit mitigation technique to incur slowdown upon the target application being protected.

- **Security Tradeoffs** Security tradeoffs specifically refer to exploit mitigation techniques that intentionally sacrifice a certain aspect of their design to instead achieve a different practical property; notably, one typical tradeoff made is that practical design features such as performance or scalability are often overlooked in order to deliver a strong level of security. Defenses that implement this strategy do raise the bar for the level of security provided somewhat, but the realistic probability of them being adopted by users is slim. In contrast, exploit mitigation designs which limit or loosen security constraints imposed risk being broken once a flaw is identified. Prime examples of this include defense techniques such as Address Space Layout Randomization (ASLR) [210] and seccomp [111, 115]. ASLR performs a single load-time randomization of the starting base address for the runtime placement of the code region of an application. Since ASLR's deployment, it has already been identified that a single memory disclosure from a memory safety vulnerability can entirely nullify the randomization imposed on the application [26, 193]. Seccomp instead tries to reduce the exposed attack surface for a given executable by disabling all unused system calls. However, it has been realized that specific system calls often targeted by attackers (*e.g.*, `execve()`, `mprotect()`, `mmap()`) cannot be disabled as they are legitimately required for process creation. When designing a new exploit mitigation, security strength should always be the primary goal to strive towards.
- **Limited Scalability** Challenges in scalability include requiring additional hardware resources such as CPU cores or allocation of large chunks of memory dedicated for an exploit mitigation's runtime. For example, some defenses [13, 87, 207] employ background monitors or virtualization to enforce security which can only scale by replicating the monitor for each application thread, thereby requiring additional threads or hardware for their protection mechanisms. This challenge also includes the limited ability to perform memory de-duplication to be able to share shared code/-data efficiently. For instance, very few fine-grained randomization approaches are capable of code sharing besides MARDU[108] and Oxymoron [13]. Other defenses regrettably increase memory use by leveraging multiple permutations of code layout loaded into runtime memory [56] or additional hardened versions of functions [150] in order to provide sufficient security.
  An exploit mitigation design with limited scalability will likely result in being a very resource-draining defense taking away valuable host system resources that could be otherwise used for useful work. Remedies for this challenge are indeed very difficult

to incorporate as early design decisions greatly dictate whether scalability is possible at all. Therefore it is up the developer to be mindful of design decisions that limit scalability and avoid them if possible.

- **Fragility** Fragility can result from several contributing factors in exploit mitigation design. Ideally, when a defense framework is deployed on a target application, original application semantics should be maintained throughout runtime, and the defense should never break the application. Many, if not all exploit mitigation designs rely on some form of analysis upon the target application to be protected. Fragility essentially stems from inadequate analysis. Regardless of whether static or dynamic analysis is utilized by a defense, each has their own merits as well as disadvantages. Static analysis approaches, as in debloating [3, 4, 125, 181, 183], can inadvertly miss valid control-flow paths or remove functions needed in other valid application configurations. Precisely deriving all necessary information from an application using static analysis, for uses such as security, is undecidable and remains an open research area [128]. In dynamic analysis, approaches rely on various metrics such as provided test cases [78, 240], runtime inputs or control-flow [62, 163, 219], or even machine learning [90], which may still not be sufficient to exercise all applicable code paths or functions. Additionally, exploit mitigation frameworks that rely on dynamic analysis to adjust security policy constraints on-the-fly can also incur performance impact on the target application, compared to static analysis which does all of its computation offline.

  Thus far, exploit mitigation designs have had to make a difficult decision regarding how much fragility their defense framework inherently possesses. Some techniques overapproximate their analysis in order to circumvent fragility. As a result, this often leads to a defense that gives an attacker just enough freedom to evade the defense. For example, in various Control Flow Integrity (CFI) techniques [1, 62, 161, 163, 219, 238], overapproximated analysis can generate code pointers who have a larger-than-necessary set of valid target functions, commonly called an equivalence class [30], instead of enforcing the one true legal target [95]. Moreover still, applying a different configuration can render a protected application inoperatable unless it is re-analyzed and re-compiled with the corresponding defense policy for that new configuration. Similarly, techniques such as debloating [3, 90, 125, 180, 181, 196], which aggressively remove code from application executables, are susceptible to breaking when an unexpected control sequence is encountered because of insufficient analysis or test cases being provided.

  Fragility is another really difficult challenge to address. In many cases fragility is unacceptable, especially for mission-critical applications. Static and dynamic analysis will not improve unless the security community is able to extract more context clues and gain further insight about expected and legitimate program semantics. Fragility can be potentially resolved with a research effort to advance the understanding of legitimate program operation. Concretely, this would invole deducing new contexts still hidden within code in order to derive new potentially helpful metadata informa-

tion to reduce fragility and coincidently strengthen security. Further research into various yet unexplored contexts, especially in general code and binary analysis, help eliminate fragility as a challenge in practical exploit mitigation design.

1.1.4 Goals

The described fundamental challenges in practical exploit mitigation design have forced techniques to choose between them or compromise in one way or another. At the same time, these challenges have prevented mainstream adoption of many proposed exploit mitigation approaches.

This dissertation seeks to address the following questions:

- What new insights could be leveraged to simultaneously advance the strength of security guarantees and enhance the understanding of practical exploit mitigation design, especially for code re-use and system call abuse?
- Is it possible to resolve the performance, security, scalability, and robustness challenges in order to achieve practical exploit mitigation design?
- If so, how? Particularly, what would a practical exploit mitigation design look like?

To resolve these questions, it is necessary to not only consider which paradigms, for instance, which code components, are essential to protect, but also which are the most effective to protect. Given that there exist several major exploit mitigation design archetypes, there still does not appear to be a clear winner. Additionally, surverying code re-use defense literature suggests that it is still unclear what the most security-critical resource or code component should be of utmost importance to safeguard.

This dissertation describes three key contributions to demonstrate that practicality is possible and should be considered more seriously in exploit mitigation design. Specifically, a broad survey regarding code re-use and current mitigation techniques is presented to establish an understanding of why practical design is hard in practice. Following, two exploit mitigation designs are explored to their full extent to display their merits; both use insights from deriving core aspects absolutely essential to fulfull each defense's protection scope.

1.2 Contributions

The overarching goal of this dissertation is to design strong exploit mitigations that are practical enough to be deployed among the general public *and* capable of denying advanced attacks. This dissertation specifically defines the following four axioms for considering an exploit mitigation design to be practical.

- **Low Performance Impact** This dissertation aims to design exploit mitigations

whose runtime overhead is acceptable by its end users. In reality, achieving a negligible overhead is challenging due to the inherent computation costs associated with tracking and decision making for verification. However, this does not mean a new exploit mitigation technique should not strive to minimize its performance impact and aptly balance practicality tradeoffs. Specifically, this dissertation aims to design practical defense techniques that have a performance overhead of no more than 10%, and ideally less than 5%.

- **Strong Security Guarantees** A practical exploit mitigation design should provide strong security guarantees so that when deployed, it sucessfully upholds its security guarantees and protects the system against all attack vectors in its defined scope. This axiom is the most important to consider in design, however it is also often the most challenging obstacle to overcome. Broadly speaking, the potency of a security policy enforced by a mitigation technique is often proportional to the performance incurred and resources required.
- **Scalability** To be considered scalable, this dissertation aims for its exploit mitigations to be cognizant of resource usage. This includes usage of additional memory and CPU cores that could otherwise be utilized for useful work by applications on the host machine. This dissertation intends for a practical defense to incur zero or minimal usage of additional system resources.
- **Robustness** This dissertation aims to design exploit mitigations that are generic. More specifically, these designs should support diverse software applications, they should not be reliant on hueristics or offline training, and they should not require special configuration. Relying on such constructs can lead to unintentional side effects, such as breaking the applications functionality, or detecting false positives. For example, some defenses, like debloating, intentionally remove code from an executable in order to reduce its attack surface. Meanwhile, defenses that require analysis can sometimes miss valid control-flow paths, leading to the defense mistakenly marking a legitimate control-flow transition as an attempted attack during runtime. This dissertation intends for practical exploit mitigation designs to not have such flaws. Upholding this axiom will additionally increase a design's attractiveness towards general adoption, especially for mission-critical applications, where robustness is expected.

This dissertation presents an in-depth survey of the current security landscape surrounding code re-use and system call abuse. Additionally, two research thrusts are presented to showcase the viability of designing practical exploit mitigation mechanisms while following the above defined axioms. The first thrust, MARDU, describes an innovative approach to resolve the chief challenges currently observed when deploying a re-randomization based defense. Randomization, especially passive randomization, simply does not offer enough entropy to be considered secure; a single memory disclosure vulnerability can foil this defense. On the other hand, active re-randomization has sufficient entropy, but is an expensive mechanism to employ; most techniques even require stopping-the-world in order to perform re-randomization and update all necessary data to maintain semantics during run-

time. The second thrust, BASTION, describes a new design reflecting the recent increased focus on system calls being rediscovered as a security-critical code component in many modern attack models. Especially recently, the innovation of system call specialization, such as `seccomp` [115], has shown this urgency to adequately protect system calls. Yet, few recently proposed approaches take the initiative to raise the bar in making system call usage more secure. Therefore, this dissertation makes the following contributions:

- A broad study of the security landscape is presented. This preface to the following two practical exploit mitigation solutions aids in advancing the current understanding of security from the perspectives of both the attacker & defender. It also serves as a summary of the current state-of-knowledge regarding code re-use. This study explains prominent attack vectors that remain at large and clearly identifies where these attack vectors strike. Moreover, it presents a taxonomy that distinguishes and explains prominent exploit mitigation techniques into several unique archetypes and subtypes. This taxonomy explains how each exploit mitigation category achieves sufficient protection as well as briefly summarizing notable works.
- The design, implementation, and evaluation of MARDU, an on-demand, scalable, re-randomization defense is presented. MARDU refines the standard randomization exploit mitigation design in several regards. MARDU realizes a competitive performance tradeoff by only re-randomizing code at program start and on-demand when suspicious attacker behavior is observed during runtime. Additionally, MARDU is capable of sharing randomized code throughout runtime. As a result, MARDU's re-randomization mechanism is scalable and supports deduplication of shared code such as linked libraries, letting it be deployed system-wide without compatibility issues. To accomplish this, the MARDU defense framework is made up of custom LLVM compiler code transformation passes as well as a modified Linux kernel.
- The design, implementation, and evaluation of BASTION, a specialized context-sensitive (fine-grained) defense for enforcement of legitimiate system call usage is presented. BASTION applies general known contexts under a new frame of reference (surrounding system calls) to significantly strengthen the security guarantees able to be employed towards protecting system calls. BASTION specifically addresses the current lack of fine-grained integrity for system calls among the security community, especially when they can be considered a security-critical code component in code re-use. BASTION introduces the new concept of system call integrity to enforce the legitimate use of system calls in a program during runtime. By concentrating its efforts to encompass system calls with three strong invariants, BASTION's performance is negligible in most cases even though it is a context-sensitive approach. Similar to MARDU, BASTION's defense framework is based upon custom LLVM compiler code analysis, minimal instrumentation, and a separate runtime monitor.

1.3 Dissertation Organization

This dissertation is organized as follows. Chapter 2 presents background information regarding this dissertation. This includes a study of the current security landscape explaining relevant attack models, code components that are traditionally targeted by attackers, and providing an informal taxonomy of modern exploit mitigation strategies that are employed to address these attacks. Chapter 3 describes MARDU, a re-randomization-based defense technique that presents solutions to challenges inherent in re-randomization defense strategies, such as unattractive performance tradeoffs and the inability to share dynamically randomized code. To the best of my knowledge, MARDU is the first defense framework capable of sharing re-randomized code system-wide throughout runtime and has a competitive performance footprint by performing randomization on-demand. This chapter presents the full design, implemenation, and evaluation details for this contribution. Thereafter, Chapter 4 describes BASTION, a specialized system call filtering mechanism aimed at significantly strengthening the security surrounding system calls. This work specifically seeks the address the current lack of specialization regarding security-critical functions such as system calls. As done for MARDU, this chapter also presents the full, implementation, and evaluation details for this contribution. Chapter 5 outlines future work and research directions that could be explored for each the contributions described in this dissertation. Chapter 6 concludes, summarizing this dissertation's overarching themes and reiterates its research vision.

Chapter 2

Background

This chapter goes over background information relevant for comprehension of this dissertation and its novel contributions. A major component relies on understanding: a) what code components are typically exposed and exploited by attackers in the current code ecosystem, b) what attack vectors are leveraged to reach and manipulate those particular code components, c) modern attack defenses that have been proposed by the security community as well as defense techniques currently adopted by main-stream computing, and d) the relationship and interconnection between attacks and defenses, specifically their history, current trajectory, and evolution to continuously outclass and overcome one another. Additionally, understanding the predominant defense archtypes and their relative strengths and weaknesses used in modern defenses can also help grasp understanding of the design decisions made for this dissertation.

Section 2.1 first briefly provides definitions of commonly targeted code components that are conventionally used to form attack chains. Section 2.2 then defines prominent attack vectors that modern defenses are expected to defend against. We specifically discuss those attacks that we believe this research scope can confidently address. Section 2.3 describes the major defense archtypes that are generally used to address modern attack vectors. Specifically, this section describes what they are, how they work, and what security benefits they provide for applications and the host platform. Additionally, notable works for each archtype are briefly described.

2.1 Attack-Targeted Code Components

This section presents a brief summary of commonly used code components in attacks. Each definition explains what they are, how they are typically used in software development, and their vulnerability when maliciously used by attackers. These components are inherent in software design and cannot simply be removed to negate an attack vector that utilizes them. These components are tightly integrated in C/C++ code development, for example, the object paradigm in C++ programming, the necessity of pointers for dynamic memory allocation. Note that for all components listed, except C++ objects which is particular to C++, are integral elements in both the C and C++ programming languages. An important point to remember here is that while these components allow simplified programming and

access to important operating system features, they can likewise be overtaken by attackers for malicious purpose.

Pointers (Both Code & Data) Pointers are variables whose value is the address of another variable, *i.e.*, a direct address of a memory location. The purpose of pointer is to save memory space and achieve faster execution time. Pointers can point to both data and function locations, and are called as such, a data pointer and function pointer, respectively. Because of the inherent flexibility of pointers, especially `void` pointers, they are a prime target to be corrupted and replaced with a malicious attacker desired variable value or function target. Such that when a corrupted function pointer is reached during program execution, it will execute the attacker directed function, rather the originally intended function.

Non-control Data Non-control data are ordinary data variables in a program that are used to guide the intended flow of program execution. As identified by Chen *et al.* [40], non-control-data are data such as configuration data, user input, user identity data, and decision-making data and are considered critical to software security. For example, this can be a variable compared in an `if-statement`, an index variable for an array, or even the input argument or flag for a function call as in system calls. At the same time, non-control data is much more inconspicuous. Particularly, these variables are not primary targets for attackers as they traditionally require significantly more effort and hoops to be jumped through in order to enable them for malicious use, but it is still possible. Challenges include the inherently short active "lifetime" of local variables on stack and potential difficulty in being reached without imposing or being imposed by side-effects within code blocks that update a particular non-control data variable. [39, 42, 215]. Instead of an attacker targeting function pointers, these variables can also be used to control program flow in malicious ways.

Code Gadgets Also known as ROP-gadgets, are not a tangible element that can be programmed by software developers. Rather, code gadgets are the byproduct of how code is compiled after it is written. Specifically, code gadgets are small exploitable code sequences of assembly code that fulfull functionality for an attacker. These typically end with an intended return instruction, but can also end with an unintended return (since x86-64 instructions are variable length and can be reinterpreted as different instructions at different offsets). Note that viable code gadgets can arise even unintentionally in binary code, particularly in the X86 and other Complex Instruction Set Computing (CISC) Instruction Set Architectures (ISA)'s, where its assembly does *not* use *fixed-width instructions* (compared prominent Reduced Instruction Set Computing (RISC) architectures like ARM, PowerPC, or RISC-V). These snippets can perform a vast array of operations such as addition, subtraction, setting of variables, etc. Given enough viable gadgets present in an application, manipulation with gadgets can even be Turing complete [193]. To use these small snippets of code, attackers will leverage stack manipulation in order to chain together various code gadgets present in the victim application to achieve their attack. The stack return addresses are corrupted (*e.g.*, via buffer overflow) in order to change the return target to the next gadget such that the code gadgets are executed in the order desired by the attacker.

C++ Objects & Virtual Function Tables Traditionally, classes in C++ are used to setup a template for a given entity. Classes act as a data type that can have multiple objects or instances of the class type while objects are used to instantiate those C++ classes. When objects are instantiated with a particular class, memory is allocated and the objects can then be used to access the data members and member functions of that class. This includes virtual functions which are a key part of dynamic polymorphism. Whenever a class defines a virtual function, an additional hidden pointer member is also created. This hidden member points to an array of pointers to all virtual functions of that class, called the virtual function table. The virtual function table entries are conventionally used at runtime in order to invoke the appropriate function implementation (*e.g.*, the base function or a derived version implemented by a child class), since this cannot be determined during compile-time. Virtual function table pointers are written only once at the object's construction time and are static for the entire lifetime of the C++ object. Critically, this hidden member reaching the virtual function table is an ordinary pointer and is not typically protected. Moreover, C++ objects can be counterfeited and injected by an attacker. As such, hijacking the virtual function table pointer of an object [31, 103, 236] or creating and injecting an attacker-controlled counterfeit object [192] are commonly exploited attacks on this code component.

System Calls System calls are a vital part of the software development ecosystem. System calls are API calls that allow user programs to interact with the operating system (OS). When invoked, they provide various vital services used in program execution. These services include process creation, memory management, accessing and performing operations on files, device management, and providing communication mechanisms as in networking. Used legitimately, system calls help manage and productively interact with the OS. Used maliciously, system calls can be leveraged to leak sensitive data and even gain control of the host machine from which the victim program was exploited on.

2.2 Prevalent Attacks

This section presents a brief summary of various attacks and attack models that this dissertation seeks to address with its research thrusts. Specifically, we present attacks that are capable of being addressed with randomization-based defenses and system call specialization. Since the advent of Data Execution Prevention (DEP), code-injection attacks, whereby an attacker created payload is directly injected into an application data section, have been eliminated from being viable in modern computing. This drove modern attacks to evolve to instead leverage various forms of code re-use. This type of attack class leverages code present in the victim application to form an attack against itself, rather than introducing new code made by the attacker themselves. While no new code is not introduced in code re-use attacks, the attacker can still take hold of and manipulate the values of variables at will in most cases. Note that this section does not contain an exhaustive list of modern-day attack models; instead only those attacks relevant to this dissertation are presented. Specifi-

cally, all attack variants listed below for the most part fall under the umbrella of code re-use. Additionally, malware is considered out of scope for this dissertation.

Return Oriented Programming (ROP) Return oriented programming is an attack technique that utilizes the concept of code gadgets (Section 2.1) in order to create an attack. By obtaining control of the call stack, the attacker can piece together and connect code gadgets within existing trusted software and manipulate control flow as desired. ROP attacks rely on identifying the location (offset) of useful code gadgets to be able maneuver control flow to reach and execute them. In the traditional model of ROP, the attacker needs to run offline analysis in order to find these useful code gadgets for their exploit. There exist a plethora of resources and tutorials regarding ROP gadgets online. This includes even real tools that are well maintained and capable of finding code gadgets in production applications, including ROPgadget [190] and Ropper [191]. Many popular applications (*e.g.*, Mozilla Firefox, NGINX, GIMP, etc.) as well as third-party code libraries (*e.g.*, Python, SQLite, OpenSSL, LibTIFF, php, etc.) are often open-source and therefore both the application executable and source code are publically available for download and use. Once all needed code gadgets are located, the attack can be carried out.

Just-In-Time ROP (JIT-ROP) Just-In-Time Code Reuse [201] works incrementally from a single code pointer vulnerability. Given a single disclosed code pointer, which reveals a valid memory location, obtained from a present memory vulnerability, JIT-ROP is then able to further leap frog and map the remaining valid memory pages. Once more code is diclosed, JIT-ROP is able to dynamically determine viable code gadgets and system calls available for use during runtime. After all this information is extracted, a payload can be dynamically generated for which a control flow vulnerability can be leveraged to direct control flow to the attackers exploit payload.

Code Inference & Blind ROP Code inference [173, 200] and Blind ROP [25] attacks infer an application code layout and contents via observing differences in execution behaviors such as timing or program output. From there a similar approach is taken to generate a payload with reachable code gadgets and pivot on a control-flow vulnerability to execute the attacker payload.

Code Pointer Offsetting If code pointers are not protected from having arithmetic operations applied to them, code can be susceptible to an attacker code pointer offsetting attack. Specifically, the low-order bytes of code pointers can be partially overwritten and leveraged, thus being redirected to another *unintended* valid relative offset within a randomized code region [227]. With the capability to redirect code pointers, other ROP gadgets can be reached and a payload can be created.

Privilege Escalation Attacks Privilege escalation attacks attempt to illegitimately gain elevated rights (*e.g.*, `root`) and privileges beyond what is intended for a given user on a host machine. If achieved, the attacker will then have illegitimate administrator access and permissions to the machine, allowing them to perform malicious actions at will. While the

attack class of privilege escalation is very broad in terms of methodology to achieve it, this dissertation specifically focuses on addressing vertical privilege escalation [151] originating from available vulnerabilities and exploits found in a victim application.

Control Data Attacks Control data attacks attempt to compromise a victim program's control-flow to be redirected to an attacker's exploit payload. This attack class directly targets code elements that are responsible for managing control-flow during a programs runtime (*e.g.*, return addresses, function pointers, an objects virtual function pointer table in C++) [40]. Function pointers are dynamically set to functions within a program during runtime, while return addresses allow a program to return to a previous caller after completing a function routine. In C++, objects are used to instantiate a class, thereby giving access to a particular classes functions. Counterfeit Object-oriented programming (COOP) [192] can be considered a specific instance of a control data attack particular to C++ that leverages injection of counterfeit C++ objects into a program whereby the counterfeit object's virtual function pointer field is manipulated to build and execute a malicious payload.

Non-Control Data Attacks Manipulating a program's data, rather than its control-flow, can be sufficient for crafting a viable exploitation of a victim program. Non-control data includes, as its name implies, data that does not influence control-flow; for example, configuration data, user input, and argument flags are considered types of non-control data. Non-control data attacks can be used to mount privilege escalation or leak sensitive information as demonstrated by Chen *et al.* [40] and Hu *et al.* [94]. Instead of attempting to attack code components that are typically protected by standard defenses (*i.e.*, control data), this attack leverages corruption of non-control data instead (Section 2.1). In practice, the amount non-control data present in any given code base greatly outnumbers the amount of control data, requiring much more coverage to guarantee adequate protection. Therefore, at time of writing, the security community still considers this an open reseach problem as current comprehensive data integrity mechanisms (*e.g.*, DFI [38]) are still impractical to be realistically deployed in production environments. Note that while non-control data attacks are considered out of scope for this dissertation, they are still are a valid concern in modern computing and could be covered by extending the work presented in this dissertation; it will not be discussed further.

2.3 Taxonomy of Modern Exploit Mitigation Techniques

This section presents a rough categorization of present day exploit mitigation techniques into a few major archtypes. We present each archetype and organize them from most well known to less commonly known. In each, we define what the archetype is, how it works, why it is a viable defense strategy, and any special notes if applicable. Additionally, a brief description of a few notable works is provided for each respective archetype.

To aid this section, we additionally provide Table 2.1, a generalized table that highlights

Table 2.1: Table highlighting contrasts in regards to the practicality axioms among the different major exploit mitigation archetypes. A few archetypes are composed of many approaches, and so, a range is used rather than specifying a single quantitative value.

Defense Type	Practicality Axiom					
	Performance	Security	Scalability			Robustness
			Code Sharing	Addt'l. Process	Addt'l. Memory	
Context-Insensitive CFI [1]	Excellent-Mod.	Poor-Mod.	✓	×	■	✓
Context-Sensitive CFI [95, 118]	Poor	Moderate	✓	×	■	✓
Data Integrity [38, 103, 126, 202]	Poor	Excellent	✓	×	■	✓
One-Time Randomization [119, 210]	Excellent	Poor	✓	×	■	✓
Continuous Randomization [41, 232]	Poor	Mod.-Excellent	×	●	■	✓
MARDU [107, 109]	Excellent	Excellent	✓	×	■	✓
Debloating [3, 8, 78, 125, 181, 183]	Excellent	Poor	×	×	■	×
Context-Insensitive System Call Filtering [57, 104, 115, 168]	Excellent	Poor	✓	×	■	✓
Context-Sensitive System Call Filtering [80, 150]	Excellent	Moderate	✓	×	■	✓
BASTION	Excellent	Excellent	✓	●	▲	✓

major differences, especially in terms of practicality, among these archetypes. Table 2.1 is not meant to be comprehensive, but rather to clearly and quickly illustrate the strengths and drawbacks associated to each exploit mitigation design. The *Performance* column denotes whether the exploit mitigation design has a practical, close to negligible performance impact **Excellent**, a moderate performance impact **Moderate (Mod.)**, or a significant, non-negligible performance overhead **Poor**. Under *Security*, a defense archetype can have either **Poor**, **Moderate**, or **Excellent** strength of security. **Poor** signifies most code re-use variants can easily bypass this archetype. **Moderate** means that some code re-use variants are blocked either because the design targets a particular attack variant, or it is generic but can still be evaded. **Excellent** indicates that the significant majority of code re-use attack variants can be blocked successfully by the given defense archetype. A defense archetype's ability to support *Code Sharing* is denoted with ✓; those who do not support code sharing are marked ×. Defense archetypes who do not require the use of additional process(es) (*Addt'l. Process*) are marked with ×. The remainder, denoted ●, do require a process to power framework components (*e.g.*, a runtime monitor, a trusted oracle). Defense archetypes who do not require or require minimal additional memory (*Add'l Memory* column) are denoted ■, archetypes who require moderate amounts of user memory are denoted ▲, and those archetypes who require a significant amount of memory are denoted ■. Under the *Robustness* column, ✓signifies that this exploit mitigation archetype's design is fundamentally sound and can be abitrarily applied to different application configurations or inputs. Conversely, ×means that this archetype is not generic to support arbitrary user inputs or configurations; caution should be used when employing this archetype.

2.3.1 Control-Flow Integrity (CFI)

Control-Flow Integrity is a runtime defense that prevents a wide variety of attacks from redirecting the flow of execution (control-flow) of the program. Specifically, CFI works by protects a program code pointers from abuse. By restricting the possible function targets a code pointer can be directed

to, CFI enforces that only legal function targets are available, otherwise CFI will stop execution. Since CFI's initial conception in 2005 [1], CFI has been subsequently refined over time and many diverse CFI policies have been presented. Policies of CFI can enforce constraints and checks on code components such as indirect call instructions, indirect jump instructions, and return instructions; policies can also be adjusted to a varying degree such as checking at all ICTs, or only on a specific subset (*e.g.*, system call or API invocations) [55] in order to lower performance tradeoffs. In theory, the perfect CFI defense needs to ensure that the final allowed control-flow graph (CFG) exactly matches the real control-flow, no more (which can still give enough wiggle room for an attacker to maneuver their malicious payload), no less (which can lead to false-positives or deviation from intended program semantics) [135, 226]. However, generating such a precise CFG is a challenging research problem.

Dimensions of CFI

- **Context-Insensitive CFI** This subtype of CFI is more performant than context-sensitive CFI and generally contributes negligible overhead. Context-insensitive CFI [1, 208] mechanisms typically rely on static and/or offline analysis (*e.g.*, pre-computed expected user inputs) of program metadata to form the valid CFG and each callsite's respective EC in order to enforce legitimate control-flow. However, it has weak security guarantees since it often suffers from being inaccurate and requiring overly conservative analysis in order to not break program semantics or induce false-positives during runtime. By applying such a conservative approach, context-insensitive CFI techniques leave a program's attack surface larger than necessary.

- **Context-Sensitive CFI** This subtype of CFI is much more secure and provides stronger security guarantees than traditional context-insensitive CFI approaches. Context-sensitive CFI mechanisms [95, 116, 118, 140, 163, 219, 220] refine the CFG used in enforcement by applying additional contextual information to aid in making a decision whether a control-transfer is legitimate or not. While leveraging new information adds significant hardening to the application by reducing the EC set of allowed destinations, this also adds significant performance overhead to these CFI mechanisms, as introducing contextual information requires additional processing time during the enforcement phase.

- **Coarse-grained CFI** This refers to CFI mechanisms that employ a more relaxed CFG during run-time enforcement that allows dozens or more legal execution paths. They may use more rudimentary (*e.g.*, static analysis, heuristics), but efficient designs [42, 170, 212, 235, 238]. For example, many coarse-grained CFI policies for handling return instructions only validate if the return address points to an instruction that immediately follows a call instruction [55].

- **Fine-grained CFI** This refers to CFI mechanisms that are traditionally more complex than their coarse-grained CFI counterparts in order to leverage and enforce a tighter CFG for a given application. This can include deriving additional contexts to be context-senstive, using dynamic-analysis [172], applying a shadow stack [32, 53], or employing field-senstive pointer analysis [134]. As a result, fine-grained CFI mechanisms often incur unacceptable performance overheads and are considered inpractical for real-world adoption.

Notable Works

- **CFI** [1] The very first publication introducing the concept of protecting the integrity of a programs control-flow. This seminal work aimed to mitigate control-flow hijacking attacks. CFI's core contribution is in its approach to constrain the possible targets of each indirect control-flow transfer (ICT) to a legitimate computed set obtained from static offline analysis of the given program's control-flow graph (CFG). During runtime, at each ICT callsite, the target destination is compared to the callsites equivalence class (EC), (*e.g.*, the list of valid targets). Thereby, if an indirect control-flow callsite attempts to transfer the PC to a function address that is not in the legitimate EC set, the program is terminated.

- **Per-Input CFI (πCFI)** [163] This CFI variant further constrains ICTs by first generating a valid static all-input CFG [161, 162]; this static CFG serves as the upper bound for what edges can be legitimately added to the CFG that is enforced during runtime. πCFI then dynamically builds a runtime per-input enforced CFG, incrementally adding edges while confirming that it stays within bounds of the static CFG. Thereby, πCFI enforces this dynamically built CFG based on user input rather than enforcing ICTs arbitrarily for all input as in the original CFI [1]. The authors particularly denote, that even for long running applications, the number of edges added in the enforced CFG stabilizes to a small subset of the total number of edges possible from the static upper bound CFG.

- **Griffin** [77] Griffin is a hardware-assisted CFI enforcement system. Specifically, this CFI approach uses a commercially available hardware feature particular to Intel CPUs, Processor Trace (Intel PT) [100]. Griffin leverages Intel PT to record the complete user-level execution of a monitored program and carry out runtime control-flow checks based on the recorded execution trace. It is designed to support multiple types of CFI policies including coarse-grained CFI policies, fine-grained CFI policies, and stateful CFI allowing for flexibility for the user to choose between performance and security. Stateful CFI policies use runtime execution state metadata provided by Intel PT, (*e.g.*, a call stack) to further constrain forward and/or backward control-flow transfers. While it is known that relying on hardware can bottleneck performance, Griffin performs both non-blocking and blocking control-flow checks to achieve better performance without sacrificing security.

- **μCFI** [95] Although context-sensitive CFI policies significantly reduce the allowed code targets for each ICT, most are unable to provide a unique valid target for each ICT. This work introduces and enforces the notion of the Unique Code Target (UCT) property for CFI. During runtime, the μCFI defense system enforces this property by only allowing the transition to one unique valid target for each invocation of an ICT instruction. Similar to Griffin [77], μCFI collects runtime information to augment the points-to analysis and constrain the set of allowed targets. However, instead of using user input data, μCFI identifies constraining data from within the program; Constraining data in μCFI is defined as any non-constant operand of an instruction that is involved in a code pointer calculation. Moreover, μCFI developed a way to efficiently pass this arbitrary constraining data from the execution to their defense system monitor to deduce the correct ICT target.

- **OS-CFI** [118] This work introduces the concept of origin-sensitivity as another viable type of contextual information to reduce the attack surface for ICT instructions in C and C++. OS-CFI defines origin-sensitivity as the origin of the code pointer called by an ICT. For C, this work defines the origin as the code location that most recently wrote the updated function address stored in the code pointer used at the given ICT. For C++, this refers to the code location where the receiving object's constructor is called. OS-CFI leverages the now defunct and discontinued (as of 2019) hardware-based bounds checking system, Intel MPX [99, 166], as a fast hashtable to efficiently store and retrieve the origin of code pointers during runtime. OS-CFI instruments the target program with inline reference monitors in order to collect and maintain contextual information. However, since the monitors use memory to store temporary data while searching the hash table, they are briefly vulnerable to race conditions. Therefore, OS-CFI utilizes transactional memory (Intel TSX [101]) to protect the integrity of its reference monitors.

2.3.2 Data Integrity

Data Integrity is a runtime technique that guarantees the integrity of data for a target program. The scope of data Integrity can range from protection of individually identified critical variables, subsets of variable types, all the way to all data (including heap) used by a program. Data Integrity works by protecting a set of data from malicious abuse and corruption by an attacker. Data integrity techniques can cover different scopes, BOGO [239] provides memory safety for all dynamically allocated data, CPI [126] guarantees the integrity of all code pointers, VIP [103] covers virtual function tables and heap metadata in addition to code pointers, and DFI [38] covers all readable and writable data in a program. In doing so, an attacker cannot corrupt or set necessary pointers and variables to the values specifically needed to carry out and complete an attack. Data Integrity is often regarded as the archetype that has the most code coverage in terms of protection, accordingly it also has some of the most prohibitive performance tradeoffs when employed [73].

Notable Works

- **Data-Flow Integrity (DFI)** [38] DFI focuses on enforcing that every use of data (both reads and writes) is legitimate during runtime. As a result, DFI is capable of defending against both control-data and non-control-data attacks. To accomplish this security policy, DFI utilizes a program's data-flow graph. A data-flow graph (DFG) can be extracted using static analysis; reaching definitions analysis [6] can be used to compute a static data-flow graph for a target program. This analysis computes the set of valid instruction locations that update a variable's definition (writes) for each use (reads) of a particular data variable and assigns a unique identifier to each definition. With this information, DFI performs instrumentation to tag each write instruction and adds verification checks at each read instruction. DFI then uses a table at runtime to keep track of the last instruction that wrote to each data's memory. Therefore, DFI will raise an exception if data-flow integrity is violated by confirming with this secure data table.

- **Code Pointer Integrity (CPI)** [126] CPI and its relaxed version Code Pointer Separation (CPS) are notable data integrity designs that focus on the protection of all code pointers in

a program in order to prevent control-flow hijacking attacks. CPI specifically covers all function pointers, return addresses, as well as data pointers used to access code pointers indirectly. CPI and CPS both use static analysis to identify the set of sensitive memory objects that must be protected in order to guarantee memory safety for code pointers. All sensitive pointers are then placed into an isolated secure memory region and can only be accessed with legitimate memory operations deduced from static analysis. Their relaxed form, CPS, instead only protects actual function pointers and ignores memory objects and data pointers used to reach function pointers. This version is more performant but offers looser security guarantees. Nonetheless, this form of pointer integrity is especially useful in situations where virtual function pointers are abundant, as in applications with many C++ objects.

- **HyperSpace (VIP)** [103] This work introduces the concept of a Value Invariant Property (VIP), a new security policy that thwarts both control-flow hijacking and heap metadata corruption attacks. VIP does this by enforcing the integrity of data values for security-sensitive data. This security policy capitalizes on the life cycle of security-sensitive data, particulary by deducing distinct states that security-sensitive data exist in. VIP claims security-sensitive data should only be altered by legitimate updates so it should thereby be immutable between two legitimate updates. Extending this insight, VIP is implemented to protect code pointers, C++ virtual function table pointers, and heap metadata from corruption. VIP uniquely offers a hybrid specialized scope for enforcing memory safety, rather than employing a full memory safety mechanism (as in DFI), or only protecting code pointers (as in CPI).

2.3.3 (Re-)Randomization

Re-randomization is a defense that makes reaching code gadgets harder to recover and/or reach, as well as making it more challenging to complete an attack without accidently breaking the application. Unlike information hiding, re-randomization solutions seek to re-randomize and invalidate any leaked information (ideally) before an attacker has a chance to use it. Thus re-randomization is also known as a leakage-resilient probabilistic defense strategy. This defense archetype specifically focuses on protection of code gadgets. It also can have varying degrees of granularity, by re-randomizing code at memory page (*e.g.*, in-place code randomization [48]), code or data regions [119], function [56, 232], basic-block [169], or assembly instruction (*e.g.*, instruction displacement, instruction set randomization [114]) granularity. Additionally, randomization can be specialized to be conducted on different pieces of information varying from the code layout [41, 224, 232], code pointer values [22], or the entire memory address space itself [82, 141]. By re-randomizing code, code components such as gadgets and function addresses can be challenging or impossible to reach, due to the unpredictability and continuous churn of the code layout throughout runtime. This dissertation discusses one such re-randomization technique in Chapter 3.

Types of Randomization

- **One-Time Randomization** Passive Randomization refers to mechanisms that only uti-

lize randomization once. This single randomization can be performed during compilation of the program into a binary, after the program binary has been created via binary rewriting [119, 124, 179], or during the initial stages of process creation before the kernel hands off control from from kernel space back to userspace [228]. Address Space Layout Randomization (ASLR) [210] is a prominent example of passive randomization that has been adopted and is currently deployed in Linux. This single-shot randomization scheme often incurs negligible overhead. Additionally it can often be performed on commodity off the shelf (COTS) binaries which traditionally have all symbol and debug information stripped in order to prevent reverse engineering of proprietary software and libraries. However, passive randomization has weak security guarantees as a single memory disclosure can reveal the entire code layout for an attacker to create a ROP attack.

- **Continuous Randomization** Active randomization refers to mechanisms that actively perform re-randomization multiple times throughout the lifetime of the application process. Re-randomization frequency is a unique adjustable tuning metric available in this defense strategy. Accordingly, various trigger mechanisms have been proposed to control the re-randomization frequency to strike a balance between security and performance. Re-Randomization can be triggered by a variety of different metrics including set-time intervals [41, 50, 232], system call history [22, 141, 224], or suspicious runtime behaviors like a crashed worker thread [107, 109]. Periodically re-randomizing code introduces the notion of active versus stale code images; the *active* code image represents the newly randomized code layout that is still unknown to the attacker, whereas a *stale* code image represents the previous randomization layout. Even if a stale code image is revealed or obtained by an attacker, it is no longer servicable and cannot be used to create a ROP attack. Constantly randomizing code also introduces a high level of code churn, and helps increase entropy. This makes the process carrying out all the steps of launching a malicious ROP chain difficult; the attacker must figure out a way to expose the active code layout, find the gadgets they need, construct the attack, and launch their payload all within the brief time-window delay before the active randomization defense performs its next randomization. In reality, active randomization also can impose performance overhead as maintaining semantics for so many moving pieces is a challenging task.

Notable Works

- **Address Space Layout Randomization (ASLR)** [210] ASLR is the seminal work that kicked off the entire archetype of randomization defense mechanisms. This simple mechanism works by randomizing the starting virtual address of the different parts of a program process - this includes the program's code region, its linked libraries as well as the stack and heap. ASLR works by breaking assumptions that developers could otherwise make about where programs and libraries would lie in memory at runtime. Before the existence of ASLR, the layout of a program (and its respective linked libraries) in memory could be assumed to be placed at same location (*e.g.*, a binary's code region always starting at address `0x00400000`), even for subsequent runs. Moreover, the mainstream transition in computing from `32-bit` to `64-bit` systems significantly increased the entropy ASLR can achieve. In a 32-bit system,

only 8 bits of entropy were available, whereas in 64-bit systems 17-19 bits of entropy are available to be randomized.

- **Isomeron** [56] This randomization scheme specifically aims to address JIT-ROP attacks. Isomeron creates and loads 2 distinct versions of the program code that have been diversified by an arbitrary into the programs virtual address space. Then during runtime, a coin is flipped at each function boundary, to decide which version of function code should actually be executed by the program. With this technique, even if the attacker knows both code version layouts, they can still not predict which code version of a function will be executed. As the length of the ROP gadget chain payload increases, the probability of guessing the correct sequence decreases exponentially.

- **Shuffler** [232] While many randomization mechanisms rely on a separate monitor process, Shuffler is able to achieve not only hardening security for the given application, but also protects itself from attack. This randomization scheme transforms all program code (including its own) using binary rewriting all code pointers into unique identifiers. Shuffler then operates by performing continuous code re-randomization at runtime, within the same address space and at the same level of privilege as the programs it defends. It configures itself to run in a separate thread and prepares newly shuffled code at each time interval, updating all code pointers and return addresses as necessary.

- **RuntimeASLR** [141] This defense aimed to address clone-probing attacks, which enable ROP. Clone-probing attacks abused the fact that newly spawned children processes via `fork()` from a parent application inherit the process state of their parents, *including randomization of the code layout*. This allowed the attacker to repeatedly probe different children clones in order to incrementally expand their knowledge of the program's memory layout to construct a malicious ROP payload. As such, RuntimeASLR devised a runtime randomization scheme that re-randomizes the address space of every child after `fork()` is called. Particularly, their approach was successful in being randomization scheme that supports code sharing and preserves semantics for multi-process and multi-threaded scenarios. Moreover, this randomization scheme only required a custom library to be implemented; it did not require an additional process to work.

- **CodeArmor** [41] In this randomization scheme, a unique code space organization and a level of indirection (via an offset) is introduced to make reaching mapped code locations challenging for attackers. CodeArmor creates a code new layout in virtual memory, Concrete Layout Independent Code (CLIC), which splits a programs code region into 2 separate regions (the virtual and concrete code code regions). This layout decouples code pointers stored in data memory from the concrete location of their targets in the address space. In order to maintain semantics and reach the real concrete code region, CodeArmor then leverages an offset hidden from the attacker to calculate the real code address target. Meanwhile direct calls and jumps in CodeArmor simply transfer control to a concrete target relative to the current (concrete) instruction pointer and thus require no virtual-to-concrete translation.

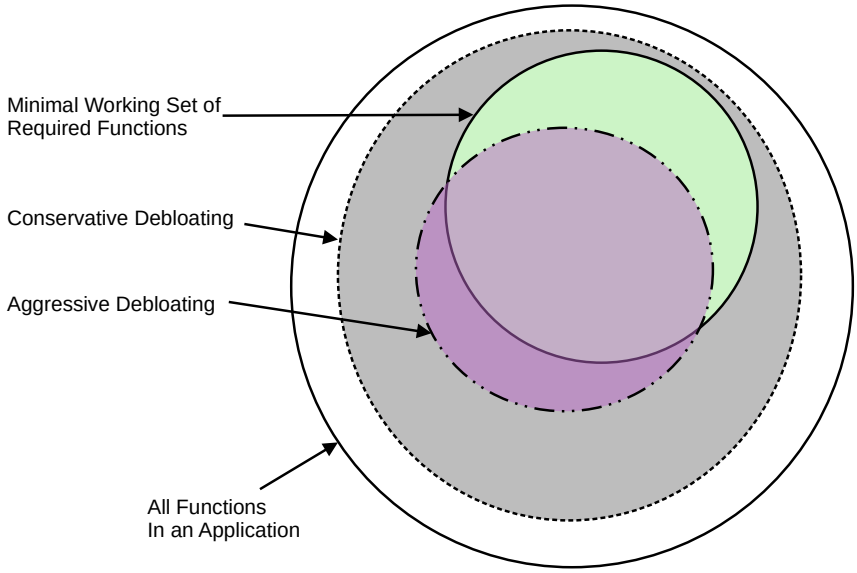


Figure 2.1: Overview of different types of debloating. This shows the visual representation of what code is potentially cut by conservative versus aggressive debloating in regards to the entire codebase and the minimal working set of functions needed to maintain semantics and not be fragile.

2.3.4 Debloating

There is circumstantial evidence that library code bloat is pervasive. For instance, over 95% of glibc code is never used by production applications [181]. To add, a very large number of ROP gadgets exist in glibc, to the point that Turing-complete and -incomplete programs can be stitched together [175, 184]. Debloating is the process of purposely removing code, specifically, dead or unwanted code, in order to reduce the viable attack surface an application exhibits in runtime memory. Debloating can be performed at various granularities (*e.g.*, file, function, basic-block scopes) and works by removing as many code chunks as possible from application code and/or its linked shared libraries. By *physically* removing code chunks, the various code gadgets embedded within program and library code are inherently removed as well, reducing their overall quantity and limiting their expressiveness (utility). This defense then potentially makes creation of certain attack vectors impossible since there do not remain enough necessary code gadgets to create the attack after debloating is performed on the target executable. However, using ROP gadget reduction as a security metric has caused major controversy in the security community because removing unused functions or unneeded features cannot prevent code-reuse attacks entirely. Debloating can be configuration-driven, as in Koo *et al.*. [125], based on code dependencies (Piece-wise [181], Nibbler [3]), deduced from call graphs and/or control-flow graphs (μTrimmer [237]), or with user test cases [78, 180]. Notably, because of this technique's dependence on accurate analysis, and the inherent physical destruction of code employed by this technique, it is possible for this approach to be fragile. Especially if analysis is incomplete, code that is needed for legitimiate execution of an application could be accidentally removed, thus breaking program semantics.

Types of Debloating

- **Conservative Debloating** This type of debloating prioritizes robustness and soundness over security in its design. Debloating mechanisms that follow conservative principles strictly remove only those functions whose removal is deemed safe. Consequently, conservative approaches leave a considerable amount of unwanted code linked to the final application binary, significantly degrading security. Piece-wise [181] is one such debloating mechanism which adheres to an assumption of soundness in its approach for generating its final set of included library functions. Nonetheless, soundness in deployed exploit mitigation designs is appreciated, especially when dealing with mission critical applications.

- **Aggressive Debloating** On the other hand, this type of debloating disregards the importance of soundness, and assumes the user can tolerate restarting the application after crashes or incorrect program execution. Prime examples include Chisel [90] and RAZOR [180]. Aggressive but unsound approaches offer significantly stronger levels of security since more code is removed, but also understandably suffer from usability issues in cases where the developer does not have a comprehensive set of test cases to exercise all possible code paths. For this reason most such proposed debloating designs are impractical, especially since brute-force is a known typical attacker strategy.

- **Constructive Debloating** A rather new development [175, 177], this type of debloating performs the inverse of traditional debloating. Traditional debloating can be considered *destructive*, where program code base is pruned and reduced to a minimal form to minimize available gadgets (making this as mentioned earlier very prone to being unsound); the remainder is made no longer usable to anyone either by being overwritten (*e.g.*, `0xCC`) or physically recompiled into a smaller binary. Instead this form of attack surface reduction is *constructive*, where all program code is initially disabled and a runtime component activates code that is necessary at any particular execution point.

Notable Works

- **Piece-wise** [181] Piece-wise improves purely static debloating techniques by collecting address-taken functions of each shared library or executable. Piece-wise is a debloating framework built on a unique two step process to remove unused functions with zero runtime slowdown. Piece-wise first uses a compiler analysis pass in order to generate a full-program dependency graph. Then, a custom loader dynamically loads the functions that are present in the dependency graph. To strip out unnecessary functions, Piece-wise's custom loader masks unused library functions when loading libraries, incuring extra load-time slowdown to use Piece-wise. Moreover, when Piece-wise is applied to multiple applications, the resulting union of all needed functions of debloated libraries will quickly lead to diminishing returns.

- **BlankIt** [175] Instead of being a traditional debloating approach that focuses on cutting & removing unwanted code, BlankIt reverses this design and instead seeks to enable "getting only what you want". BlankIt is designed to activate and de-activate library functions on-demand to keep a library's dynamically linked code attack surface to a minimum. More specifically, BlankIt loads only the set of library functions that are needed by each library callsite at runtime. This set of functions is then unloaded and disabled again once the library

function callsite returns. To make this dynamic code activation possible, BlankIt relies on an input-aware runtime oracle that predicts the set of library functions needed. BlankIt's just-in-time loading oracle is implemented using Intel Pin [142]; to operate BlankIt, the target application's source code and sample inputs are required in order to train the decision-tree based call path predictor. As a result, BlankIt's aggressive dynamic debloating strategy shows a very high percentage of code reduction, as only a small portion of library functions are visible during any given runtime window.

- **μTrimmer** [237] This debloating technique is specifically catered towards resolving key challenges that arise when attempting to apply debloating on application and library code that is destined to be deployed on embedded systems. μTrimmer particularly supports use cases where no source code or debug symbols are provided, as in stripped binaries. μTrimmer is a input-profile-agnostic, static library debloating technique for the MIPS architecture. μTrimmer offers a number of key insights for deriving a tighter and more robust debloating mechanism. First, μTrimmer performs debloating at the granularity of basic blocks (rather than functions) to sidestep obscure function boundaries caused by multi-entry functions and tail call optimization [146]. Second, μTrimmer leverages the MIPS Application Binary Interface (ABI) [211] to derive hints that further improve address-taken blocks/functions detection process. Additionally, μTrimmer reads the applications global offset table (GOT) to deduce address loading patterns of the GOT and decide all legitimate addresses that could be referenced. All of these insights allow μTrimmer to generate an inter-procedural control-flow graph (ICFG) much more accurately and for each library compared to predecessors.

2.3.5 System Call Filtering

System Call Filtering is a light-weight defense that focuses on security critical functions, specifically system calls. System Call filtering works by performing analysis on an application to derive the subset of system calls necessary for its normal operation. This defense then creates a policy to be applied at the start of program execution for the given application which specifies the system calls that are enabled for use and disables all other system calls. System call filtering essentially reduces the total kernel surface exposed to untrusted applications with negligible performance impact. By limiting the amount of system call APIs available to only those essential for application operation, this defense constrains the available tools (*e.g.*, system calls) for an attacker to leverage when attempting to hijack a victim application. Seccomp [111, 115] is a prominent system call filtering feature that is deployed on Linux and available for users to integrate into their applications. Additionally, this dissertation introduces BASTION, a design for a new approach to achieve more complete system call filtering in Chapter 4.

Types of System Call Filtering

- **Context-Insensitive Filtering** This type of system call filtering does not leverage additional contexts whatsoever in its exploit mitigation design. As a result, the effective coverage of context-insensitive system call filtering is limited. Specifically, the security guarantees remain the same as ones outlined by seccomp itself. Moreover, a large majority of system

call filtering mechanisms are still focusing on addressing accessibility and portability issues associated with seccomp for the general public rather than strengthening its inherently rudimentary security guarantees. To illustrate, most techniques proposed thus far in literature such as Confine [79], sysfilter [57], and ABHAYA [168], provide automation in the derivation of viable system call filtering policies for users.

- **Context-Sensitive Filtering** Also known as system call specialization, this type of filtering aims to make use of invariants surrounding system calls in order to restrict the usage of system calls in malicious manners. Notably, very few system call filtering mechanisms in current literature have thus far explored utilizing additional contexts to strengthen security and enforce legitimate system call usage in applications. Feasible contexts that can aid in constraining system call usage include protecting the arguments for a given system call, conntecting the callsite location with the system call itself, denoting the distinct phase of execution a program is in, or leveraging other application runtime state information at the time of a system call invocation. One example of context-senstive filtering is Temporal system call specialization [80], which enforces policies that only allow certain system calls during certain phases of execution. In Chapter 4, this dissertation demonstrates substantial advances in achieving a strong and practical system call filtering mechanism.

Notable Works

- **seccomp** [104, 111, 115] This is a Linux kernel security feature first integrated into the Linux kernel's mainline codebase in 2005. At that time, it was a simple security primitive that enabled or disabled a process to be able to inoke the following four system calls: `read()`, `write()`, `exit()`, and `sigreturn()`. Since then it has evolved in complexity and even received its own system call `seccomp()` in 2014 (Linux kernel v3.17). Today, seccomp is supported by the Berkeley Packet Filter (BPF) language, enabling processes to specify which system calls are permitted, which system calls are restricted, and which system calls must specify certain argument values. However, note that these seccomp BPF filters must be manually hand-written and customized per-application and per-configuration. Moreover, seccomp only supports the enforcement of static argument values, making seccomp still be considered a coarse-grained approach.

- **`sysfilter`** [57] To resolve the manual intervention needed to apply seccomp to production applications `sysfilter` is a system call filtering framework that automates the process of deriving the minimal set of system calls needed by an application. `sysfilter` argues that without any mechanism in place to enforce the principle of least privilege with respect to the system call API, the OS kernel provides full and unrestricted access to the entirety set of system calls to any application. Therefore, sysfilter performs a two step process of system call set extraction and enforcement to provide system call filtering. With a target application binary, `sysfilter` extraction automatically resolves dependencies to shared libraries refering its ELF metadata [64], and constructs a tight but over-approximated function-call graph to extract its required system calls. With this valid set of system calls, the `sysfilter` enforcement tool automatically converts this information into a BPF program to be used with seccomp-BPF and applied to the target application. Finally, `sysfilter` automaticatically

injects the compiled filter during the initialization of the applications executable leveraging seccomp semantics and makes the respective filter visible to all executing threads. `sysfilter` is a great solution for developers looking to employ seccomp lacking the technical know-how to write their own manual filters for their applications. However, one serious drawback is that `sysfilter` is very coarse-grained. Note that ld.so is included in the analysis of identifying required system calls, as well as initialization of additional libraries at run-time (e.g., `dlopen()`). As a result, a subset of of system calls that is exclusively needed for process initialization must be included in `sysfilter`'s final valid system call set, even if the application itself never calls these system calls.

- **Temporal System Call Specialization** [80] Seeing that seccomp filters remain static for the entire lifetime of an application, this system call filtering mechanism adds the dimension of time to seccomp in the form of temporal specialization. The authors of this work noticed that certain applications, especially server applications, can be split up into different distinct phases of execution; specifically, server applications typically exhibit two distinct initialization and serving phases. Correspondingly, the application's requirements change throughout its execution lifetime. There this mechanism works by essentially limiting the set of system calls that are available to be called and further reducing the attack surface depending on which phase of execution the application is currently in. To facilitate multiple phases of execution with independent system call permissions, temporal system call specialization uses manual developer-supplied markers that indicate the beginning of the initialization and serving phases. Manual annotation is acceptable in this case given that server applications have such few distinct execution phases. Additionally, this filtering mechanism leverages the unique design of seccomp, in that seccomp is able to be appended without issue to previous seccomp filters as long as that filter is more strict. Note that seccomp only works in one direction and does not support appending filters which are more relaxed.

- **Saffire** [150] Saffire is one of the first instances of context-senstive system call filtering. Saffire's goal is to create specialized versions of sensitive system calls distinctly tied to their callsite locations in order to restrict how system calls are invoked. This goal is derived from observation of code re-use attack patterns and discerning that system calls made by an application are semantically different from malacious invocations made by an attacker. To achieve this goal, Saffire defines a system call's calling context as the callsite locations where it can be legitimately invoked. Then for each calling context, Saffire creates customized function copys that constrain the arguments to either expected values. For static values, Saffire can easily hard code the expected argument input. On the other hand, Saffire handles dynamic values by tagging the last legitimiate update of the argument variable. This static and dynamic argument binding is a key contribution in defining contexts for system call filtering, but it could still be significantly improved. In cases where the sensitive system call is invoked with multiple different static values, Saffire then resorts to using a list or range to overapproximate the static argument inputs to not break program semantics. Moreover, Saffire's dynamic argument binding verifies dynamic values with the closest respective write location for a given argument; there are attack vectors possible where an argument can be corrupted much earlier in program flow meaning Saffire would completely miss this attack vector. Finally, Saffire is not a sufficient stand-alone exploit mitigation design. It aims to

compliment and further strengthen security with another deployed defense mechanism such as CFI.

2.3.6 Information Hiding

Information Hiding is the process of effectively hiding or making runtime memory regions inaccessible to illegitimate access, for both read and write operations and aims to be leakage-resistant, compared to re-randomization which aims to be leakage-resilient. This archetype extends the well known Address Space Layout Randomization (ASLR) [210] and similar finer-grained variants [91, 119, 169]. While the process of memory reading and writing is an inherent operation during program runtime, information hiding specifically aims to prevent disclosing the locations of sensitive code components like gadgets. Leveraging eXecute-Only Memory (XoM) or similar semantics for code pages [81, 174, 207, 231] and/or code pointer tables (*e.g.*, the Global Offset Table (GOT)) [49], is another way code can be "hidden" from arbitrary memory reads by an attacker. Thus, this defense acts to cut access from the memory region for attackers. Or to at least make it challenging for attackers to determine to memory layout in order to derive viable code components, similar to re-randomization. Note that information hiding is considered out of scope for this dissertation and will not be discussed further.

2.3.7 Memory Safety

Memory Safety effectively protects memory from dangerous security vulnerabilities and unintentional software bugs. Memory safety works by providing additional checks and performing memory cleanup for common bugs such as buffer overflows and dangling pointers, respectively. Memory safety is a needed defense paradigm as C and C++ are inherently not memory-safe programming languages. Therefore, techniques have been proposed that go after nullifying dangling pointers to prevent use-after-free vulnerabilities [54, 132, 218] provide memory safety via secure heap allocation [65, 198], and enforce bounds-checking that leverage spatial or temporal constraints [59, 63, 152, 239]. Note that memory safety is considered out of scope for this thesis proposal and will not be discussed further.

Chapter 3

MARDU: On-Demand, Shared, and Scalable Code Re-randomization

3.1 Introduction

Code reuse attacks have grown in depth and complexity to circumvent early defense techniques like Address Space Layout Randomization (ASLR) [210]. Examples like *return-oriented programing (ROP)* and ret-into-libc [193], utilize a victim's code against itself. ROP leverages innocent code snippets, *i.e.*, *gadgets*, to construct malicious payloads. Reaching this essential gadget commodity versus defending it from being exploited has made an arms race between attackers and defenders. Both coarse- and fine-grained ASLR, while light-weight, are vulnerable to attack. Whether an entire code region or basic block layout is randomized in memory, a single memory disclosure can result in exposing the entire code layout, regardless [201]. Execute-only memory (XoM) was introduced to prevent direct memory disclosures, by enabling memory regions to be marked with execute-only permissions [46, 49]. However, code inference attacks, a code reuse attack variant that works by indirectly observing and deducing information about the code layout circumvented these limitations [25]. Various attack angles have revealed that one-time randomization is simply not sufficient, and that stronger adversaries remain to be dealt with efficiently and securely [22, 141].

This fostered the next generation of defenses, such as Shuffler [232] and CodeArmor [41], which introduced continuous runtime re-randomization to strengthen security guarantees. However, these techniques are not designed for a system's scalability in mind. Despite being performant, non-intrusive, and easily deployable, they rely on background threads to run re-randomization or approximated use of timing thresholds to proactively secure vulnerable code. Specifically, these mechanisms consume valuable system resources even when not under attack (e.g., consuming CPU time per each threshold for re-randomization), leaving less compute power for the task at hand. Additionally, no continuous re-randomization techniques currently support code sharing, and thereby, use much more physical memory by countering the operating systems memory deduplication technique of using a page cache [13]. Under these defenses, the resource budget required for running an application is much higher than is traditionally expected, making active randomization techniques not scalable for general multi-programming computing.

Control Flow Integrity (CFI) is another protection technique that guards againt an attacker attempting to subvert a program's control flow. CFI in general is already widely deployed on Windows, Android [213], and iOS as well as having support in compilers like LLVM [208] and `gcc` [212]. CFI enforces the integrity of a program's control flow based off a constructed control flow graph

(CFG) as well as derived equivalence classes for each forward-edge. However, building a fully precise CFG for enforcing CFI is challenging and is still considered an open problem. Going in an orthogonal direction to CFI could avoid these inherent challenges.

In this paper, we introduce MARDU to refocus the defense technique design, showing that it is possible to embrace core fundamentals of both performance and scalability, while ensuring comprehensive security guarantees.

MARDU builds on the insight that thresholds, like time intervals [41, 232] or the amount of leaked data [224], are a security loophole and a performance shackle in re-randomization; MARDU does not rely on a threshold at all in its design. Instead MARDU only activates re-randomization when necessary. MARDU takes advantage of an event trigger design and acts on permissions violations of Intel Memory Protection Keys (MPK) [97]. Using Intel MPK, MARDU provides efficient XoM protection against *both* variations of remote and local JIT-ROP. With XoM in place, MARDU leverages Readactor's [46, 47] immutable trampoline idea; such that, while trampolines are not re-randomized, they are protected from read access and effectively decouple function entry points from function bodies, making it impossible for an attacker to infer and obtain ROP gadgets.

It is crucial to note that few re-randomization techniques factor in the overall scalability of their approach. Support for *code sharing* is very challenging especially for randomization-based techniques. This is because applying re-randomization per process counters the principles of memory deduplication. Additionally, the prevalence of multi-core has excused the reliance on *per-process background threads* dedicated to performing compute-extensive re-randomization processes; even if recent defenses have gained some ground in the arms race, most still lack effective comprehensiveness in security for the system resource demands they require in return (both CPU and memory). MARDU balances performance and scalability by not requiring expensive code pointer tracking and patching. Furthermore, MARDU does not incur a significant overhead from continuous re-randomization triggered by conservative time intervals or benign I/O system calls as in Shuffler [232] and ReRanz [224], respectively. Finally, MARDU is designed to both support code sharing and not require the use of any additional system resources (*e.g.*, background threads as used in numerous works [22, 41, 74, 224, 232]). To summarize, we make the following contributions:

- **ROP attack & defense analysis.** Our background in Section 3.2, describes four prevalent ROP attacks that challenge current works, including JIT-ROP, code-inference, low-profile, and code pointer offsetting attacks. With this, we classify and exhibit current state-of-the-art defenses standings on three fronts: security, performance, and scalability. Our findings show most defenses are not as secure or as practical as expected against current ROP attack variants.
- MARDU **defense framework.** We present the design of MARDU in Section 3.4, a comprehensive ROP defense technique capable of addressing most popular and known ROP attacks, excluding only full-function code reuse attacks.
- **Scalability and shared code support.** To the best of our knowledge, MARDU is the first framework capable of re-randomizing shared code throughout runtime. MARDU creates its own calling convention to both leverage a shadow stack and minimize overhead of pointer tracking. This calling convention also enables shared code (*e.g.*, libraries) to be re-randomized by any host process and maintain security integrity for the rest of the entire

system.

- **Evaluation & prototype.** We have built a prototype of MARDU and evaluated it in Section 3.6 with both compute-intensive benchmarks and real-world applications.

3.2 Code Layout (Re-)Randomization

In this section, we present a background on the code re-use attack and defense arms race. In summary, Table 3.1 illustrates the characteristics of each defense technique by their randomization category, attack resilience, and performance and scalability factors, and we describe these in detail in the following.

3.2.1 Attacks against Load-time Randomization

Load-time Randomization without XoM. Code layout randomization techniques such as coarse-grained ASLR [210] and fine-grained ASLR [13, 45, 56, 91, 92, 119, 123, 169, 228] which depend on the granularity of layout randomization, fall into this category of code layout randomization. These techniques randomize the code layout only once, usually when loaded into memory, and its layout never changes thereafter during the lifetime of the program.

A1: Just-in-time ROP (JIT-ROP). An attacker with an arbitrary memory read capability may launch JIT-ROP [201] by interactively performing memory reads to disclose one code pointer. This disclosure can be used to then leap frog and further disclose other addresses to ultimately learn the entire code contents in memory. Any load-time code randomization technique that does not protect code from read access including fine-grained ASLR techniques is susceptible to this attack.

Load-time Randomization with XoM. In response to A1 (JIT-ROP), several works protect code from read access via destructive read memory [207, 231] or execute-only memory (XoM) [14, 27, 41, 46, 49, 81, 174, 207, 231]. By destroying, purposely corrupting code read by attackers, or fundamentally removing read permissions from the code area, respectively, these techniques prevent attackers from gaining knowledge about the code contents, nullifying A1.

A2: Blind ROP (BROP) and code inference attacks. Even with XoM, load-time randomizations still are susceptible to BROP [25] or other inference attacks [173, 200]. BROP infers code contents via observing differences in execution behaviors such as timing or program output while other attacks [173, 200] defeat destructive code read defenses [207, 231] by weaponizing code contents from only a small fraction of a code read. Therefore, maintaining a fixed layout over crash-probing or read access to code allows inferring code contents indirectly, letting attackers still learn the code layout.

Table 3.1: Classifications of ASLR-based code-reuse defenses. Gray highlighting emphasizes the attack (*A1*-*A4*) that largely invalidated each type of defense. ● indicates the attack is blocked by the defense (attack-resistant). × indicates the defense is vulnerable to that attack. ▲ indicates the attack is not blocked but is still mitigated by the defense (attack-resilient). ✓ indicates the defense meets performance/scalability requirements. × indicates the defense is unable to meet performance/scalability requirements. **N/A** in column *A3* indicates that the attack is not applicable to the defense due to lack of re-randomization; **N/T** in column *Performance* indicates that either SPEC CPU2006 or `perlbench` is not tested. Specifically in column *A1*, ▲ indicates that the defense cannot prevent the JIT-ROP attack within the application boundary that does not use system calls; in column *A4*, × indicates that an attack may reuse both ROP gadgets and entire functions while ▲ indicates that an attack can only reuse entire functions. † Note that in TASR, the baseline is a binary compiled with `-Og`, necessary to correctly track code pointers. Previous work [224, 232] reported performance overhead of TASR using regular optimization (`-O2`) binary is ≈30-50%. MARDU provides strong security guarantees with competitive performance overhead and good system-wide scalability compared to existing re-randomization approaches.

Types	Defenses	Security					Performance			Scalability	
		Gran.	A1	A2	A3	A4	Perf.	Avg.	Worst	Code Sharing	No Addi. Process
Load-time ASLR	Fine-ASLR [45, 91, 92, 119, 123, 169, 228]	Fine	×	×	N/A	×	✓	0.4%	6.4%	×	✓
	Oxymoron [13]	Coarse	×	×	N/A	×	✓	2.7%	11%	✓	✓
	Pagerando [48]	Coarse	×	×	N/A	×	✓	1.09%	6.5%	✓	✓
	Isomeron [56]	Fine	●	●	N/A	▲	×	19%	42%	×	×
Load-time+XoM	Readactor/Readactor++ [46, 49]	Fine	●	×	N/A	▲	✓	8.4%	25%	×	✓
	LR 2 [27]	Fine	●	×	N/A	▲	✓	6.6%	18%	×	✓
	kR^X [174]	Fine	●	×	N/A	▲	✓	2.32%	12.1%	×	✓
Re-randomization	RuntimeASLR [141]	Coarse	×	●	N/A	×	×	N/T	N/T	✓	✓
	TASR [22]	Coarse	▲	●	×	×	×	2.1%†	10.1%†	×	×
	ReRanz [224]	Fine	▲	●	×	▲	✓	5.3%	14.4%	×	×
	Shuffler [232]	Fine	●	●	×	▲	×	14.9%	40%	×	×
	CodeArmor [41]	Coarse	●	●	×	×	×	3.2%	55%	×	×
Our Approach	MARDU	Fine	●	●	●	▲	✓	5.5%	18.3%	✓	✓

3.2.2 Defeating A1/A2 via Continuous Re-randomization

Continuous re-randomization techniques [22, 41, 74, 82, 141, 224, 232] aim to defeat A1 and A2 by continuously shuffling code (and data) layouts at runtime to make information leaks or code probing done before shuffling useless. To illustrate the internals of re-randomization techniques, we describe the core design elements of re-randomization by categorizing them into two types: *1) Re-randomization triggering condition* and *2) Code pointer semantics*.

By re-randomization triggering condition:

- **Timing:** Techniques [41, 232] shuffle the layout periodically by setting a timing window. For example, Shuffler [232] triggers re-randomization every 50 msec (< network latency) to counter remote attackers, and CodeArmor [41] can set re-randomization period as low as 55 μsec.
- **System-call history:** Techniques [22, 141, 224] shuffle the layout based on the history of the program's previous system call invocations, *e.g.*, invoking `fork()` (vulnerable to BROP) [141] or when `write()` (leak) is followed by `read()` (exploit) [22, 224].

By code pointer semantics:

- **Code address as code pointer:** Techniques (ASR3 [82] and TASR [22]) use actual code address as code pointers. In this case, leaking a code pointer lets the attacker have knowledge about an actual code address. Therefore, this design requires tracking of all code pointers (or all pointers) at runtime, which is computationally expensive, to update values after randomizing the code layout.
- **Function trampoline address as code pointer:** These techniques store an indirect index, for instance, a function table index (Shuffler [232]) or the address of a function trampoline (ReRanz [224]), as code pointers to avoid expensive pointer tracking. After re-randomization, the techniques only need to update the function table while all code pointers remain immutable. With this design, leaking a code pointer will tell the attacker about the function index in the table or trampoline but not about the code layout; however, because the function index is immutable across re-randomization, attackers may re-use leaked function indices.
- **An offset to the code address as code pointer:** This design also avoids pointer tracking by having an immutable offset from the random version address for referring to a function, as in CodeArmor [41]. The re-randomization is efficient because it only requires randomizing the version base address, and does not require any update of pointers. With this design, leaking a code pointer will only tell the attacker about the offset to select a specific function; however, the offset is immutable across re-randomization, so attackers may re-use leaked function offsets.

3.2.3 Attacks against Continuous Re-randomization

Based on our analysis of continuous re-randomization techniques, we define two attacks (A3 and A4) against them.

A3: Low-profile JIT-ROP. This attack class does not trigger re-randomization, either by completing the attack within a defense's pre-defined randomization time interval or without involving any I/O system call invocations. Existing defenses utilize one of timing [41, 74, 82, 232], amount of transmitted data by output system calls [224], or I/O system call boundary [22] as a trigger for layout re-randomization. Therefore attacks within the application boundary, such as code-reuse attacks in Javascript engines of web browsers where both information-leak followed by control-flow hijacking attack may complete faster than the re-randomization timing threshold (*e.g.*, < 50 msec) or not interact with any I/O system calls (*e.g.*, leaking pointers via type-confusion vulnerabilities). This bypasses these triggering conditions, leaving the code layout unchanged within the given interval and vulnerable to JIT-ROP.

A4: Code pointer offsetting. Even with re-randomization, techniques might be susceptible to a code pointer offsetting attack if code pointers are not protected from having arithmetic operations applied by attackers [22, 41]. An attacker may trigger a vulnerability to apply arithmetic operations to an existing code pointer. Particularly, in techniques that directly use a code address [22] or a code offset [41], the target could be even a ROP gadget if the attacker knows the gadgets offset beforehand. Ward *et al.* [227] has demonstrated that this attack is possible against TASR. A4 shows that maintaining a fixed code layout across re-randomizations and not protecting code pointers lets attackers perform arithmetic operations over pointers, allowing access to other ROP gadgets.

3.3 Threat Model and Assumptions

MARDU's threat model follows that of similar re-randomization works [22, 41, 232]. We assume attackers can perform arbitrary read/write by exploiting software vulnerabilities in the victim program. We also assume all attack attempts run in a local machine such that attacks may be performed any number of times within a short time period (*e.g.*, within a millisecond).

Our trusted computing base includes the OS kernel, the loading/linking process such that attackers cannot intervene to perform any attack before a program starts, and that system userspace does not have any memory region that is both writable and executable or both readable and executable (e.g., DEP (W⊕X) and XoM (R⊕X) are enabled). We assume all hardware is trusted and attackers do not have physical access. This includes trusting Intel Memory Protection Keys (MPK) [97], a mechanism that provides XoM.

MARDU does not support native MPK applications that directly use `wrpkru` instructions. We further analyze the security of leveraging protection keys for userspace in Section 3.7. Finally, hardware attacks (*e.g.*, side-channel attacks, Spectre [121], Meltdown [138]) are out of scope.

3.4 MARDU Design

We begin with the design overview of MARDU in § 3.4.1 and then go into further detail of the MARDU compiler in § 3.4.2 and kernel in § 3.4.3.

3.4.1 Overview

This section presents the overview of MARDU, along with its design goals, challenges, and outlines its architecture.

Goals

Our goal in designing MARDU is to shore up the current state-of-the-art to enable a practical code randomization. More specifically, our design goals are as follows:

Scalability. Most proposed exploit mitigation mechanisms overlook the impact of required additional system resources, such as memory or CPU usage, which we consider a scalability factor. This is crucial for applying a defense system-wide, and is even more critical when deploying the defense in pay-as-you-go pricing on the Cloud. Oxymoron [13] and PageRando [48] are the only defenses, to our knowledge, that allow code sharing of randomized code. No other *re*-randomization defenses support code sharing thus they require significantly more memory. Additionally, most re-randomization defenses [41, 224, 232] require per-process background threads, which not only cause additional CPU usage but also contention with the application process. As a result, approaches requiring per-process background threads show significant performance overhead as the number of processes increases. Therefore, to apply MARDU system-wide, we design MARDU to not require significant additional system resources, for instance, additional processes/threads or significant additional memory.

Performance. Many prior approaches [41, 46, 49, 232] demonstrate decent runtime performance on average (<10%, *e.g.*, <3.2% in CodeArmor); however, they also show corner cases that are remarkably slow (*i.e.*, >55%, see *Worst* column in Table 3.1). We design MARDU to be competitive with prior code randomization approaches in terms of performance overhead in order to show the security benefits MARDU provides are worth the minor tradeoff. In particular, we aim to ensure that MARDU's performance is acceptable even in its worst case outliers across a variety of application types.

Security. No prior solutions provide a comprehensive defense against existing attacks (see Section 3.2). Systems with only load-time ASLR are susceptible to code leaks (A1) and code inference (A2). Systems applying re-randomization are still susceptible to low-profile attacks (A3) and code pointer offsetting attacks (A4). MARDU aims to either defeat or significantly limit the capability of attackers to launch code-reuse attacks spanning from A1 to A4 to provide a best-effort security against known existing attacks.

Challenges

Naïvely combining the best existing defense techniques is simply not possible due to conflicts in their requirements. These are the challenges MARDU addresses.

Tradeoffs in security, performance, and scalability. An example of the tradeoff between security and performance is having fine-grain ASLR with re-randomization. Although such an

approach can defeat code pointer offsetting (A4), systems cannot apply such protection because re-randomization must finish quickly to meet performance goals to also defeat low-profile attacks (A3). An example of the tradeoff between scalability and performance is having a dedicated process/thread for running the defense and performing the re-randomization. Usage of a background thread results in a drawback in scalability by occupying a user's CPU core, which can no longer be used for useful user computation. This trade off is exaggerated even more by systems that require one-to-one matching of a background randomization thread per worker thread. Therefore, a good design must find a breakthrough to meet *all* of aforementioned goals.

Conflict in code-diversification vs. code-sharing. Layout re-randomization requires diversification of code layout per-process, and this affects the availability of code-sharing. The status quo is that code sharing cannot be applied to any existing re-randomization approaches, making defenses unable to scale to protect many-process applications. Although Oxymoron [13] enables both diversification and sharing of code, it does not consider re-randomization, nor use a sufficient randomization granularity (page-level).

Architecture

We design MARDU to gain insight on how to properly balance and integrate opposing goals of security, scalability, and performance together. We introduce our approach for satisfying each aspect below:

Scalability: Sharing randomized code. MARDU manages the cache of randomized code in the kernel, making it capable of being mapped to multiple userspace processes, not readable from userspace, and not requiring any additional memory.

Scalability: System-wide re-randomization. Since code is shared between processes in MARDU, per-process randomization, which is CPU intensive, is not required; rather a single process randomization is sufficient for the *entire* system. For example, if a worker process of NGINX server crashes, it re-randomizes upon exit all associated mapped executables (*e.g.*, `libc.so` of all processes, and all other NGINX workers).

Scalability: On-demand re-randomization. MARDU re-randomizes code only when suspicious activity is detected. This design is advantageous because MARDU does not rely on per-process background threads nor a re-randomization interval unlike prior re-randomization approaches. Particularly, MARDU re-randomization is performed in the context of a crashing process, thereby not affecting the performance of other running processes.

Performance: Immutable code pointers. The described design decisions for scalability also help reduce performance overhead. MARDU neither tracks nor encrypts code pointers so code pointers are not mutated upon re-randomization. While this design choice minimizes performance overhead, other security features (*e.g.*, XoM, trampoline, and shadow stack) in MARDU ensure a comprehensive ROP defense.

Security: Detecting suspicious activities. MARDU considers any process crash or code probing attempt as a suspicious activity. MARDU's use of XoM makes any code probing at-

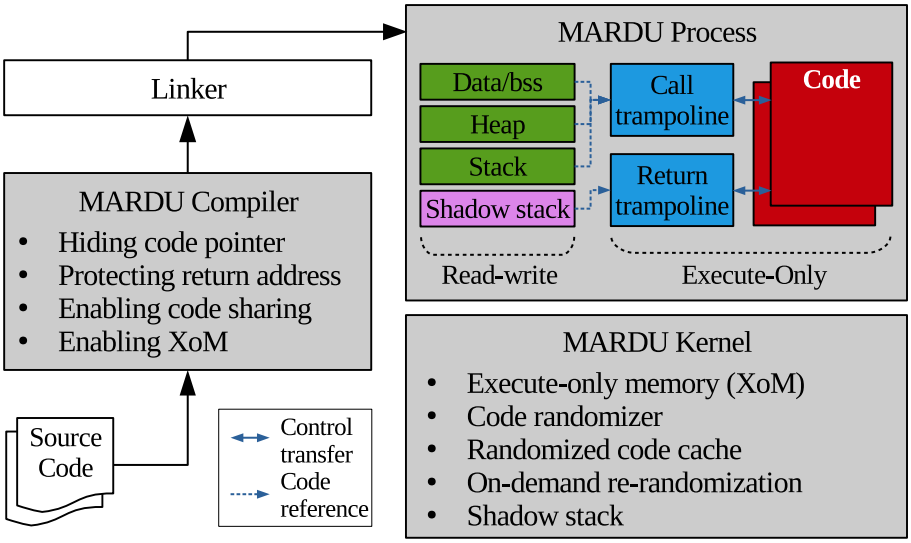


Figure 3.1: Overview of MARDU

tempt trigger a process crash and consequently system-wide re-randomization. Therefore, MARDU counters direct memory disclosure attacks as well as code inference attacks requiring initial code probing [173, 200]. We use Intel MPK [97] to implement XoM for MARDU; leveraging MPK makes hiding, protecting, and legitimately using code much more efficient and simple with page-permissions compared to virtualization-based designs that require nested address translation during runtime.

Security: Preventing code & code pointer leakage. In addition to system-wide re-randomization, MARDU minimizes the leakage of code and code pointers. Besides XoM, we use three techniques. First, MARDU applications always go through a trampoline region to enter into or return from a function. Thus, only trampoline addresses are stored in memory (*e.g.*, stack and heap) while non-trampoline code pointers remain hidden. MARDU does not randomize the trampoline region so that tracking and patching are not needed upon re-randomization. Second, MARDU performs fine-grained function-level randomization within an executable (*e.g.*, `libc.so`) to completely disconnect any correlation between trampoline addresses and code addresses. This provides high entropy (*i.e.*, roughly $n!$ where n is the number of functions). Also, unlike re-randomization approaches that rely on shifting code base addresses [22, 41, 141], MARDU is not susceptible to code pointer offsetting attacks (A4). Finally, MARDU stores return addresses–precisely, trampoline addresses for return–in a shadow stack. This design makes stack pivoting practically infeasible.

Design overview. As shown in Figure 3.1, MARDU is composed of compiler and kernel components. The MARDU compiler enables trampolines and a shadow stack to be used. The MARDU compiler generates PC-relative code so that randomized code can be shared by multiple processes. Also, the compiler generates and attaches additional metadata to binaries for efficient patching.

The MARDU kernel is responsible for choreographing the runtime when a MARDU enabled executable is launched. The kernel extracts and loads MARDU metadata into a cache to be shared by multiple processes. This metadata is used for first load-time randomization as well as re-

randomization. The randomized code is cached and shared by multiple processes; while allowing sharing, each process will get a different random virtual address space for the shared code. The MARDU kernel prevents read operations of the code region, including the trampoline region, using XoM such that trampoline addresses do not leak information about non-trampoline code. Whenever a process crashes (*e.g.*, XoM violation), the MARDU kernel re-randomizes all associated shared code such that all relevant processes are re-randomized to thwart an attacker's knowledge immediately.

3.4.2 MARDU Compiler

The MARDU compiler generates a binary able to 1) hide its code pointers, 2) share its randomized code among processes, and 3) run under XoM. MARDU uses its own calling convention using a trampoline region and shadow stack.

Code Pointer Hiding

Trampoline. MARDU hides code pointers without paying for costly runtime code pointer tracking. The key idea for enabling this is to split a binary into two regions in process memory: *trampoline* and *code* regions (as shown in Figure 3.2 and Figure 3.3). A trampoline is an intermediary call site that moves control flow securely to/from a function body, protecting the XoM hidden code region. There are two kinds of trampolines: call and return trampolines. A *call trampoline* is responsible for forwarding control flow from an instrumented call to the *code region* function entry, while a *return trampoline* is responsible for returning control flow semantically to the caller. Each function has one call trampoline to its function entry, and each call site has one return trampoline returning to the following instruction of the caller. Since trampolines are stationary, MARDU does not need to track code pointers upon re-randomization because only stationary call trampoline addresses are exposed to memory.

Shadow stack. Unlike the x86 calling convention using `call`/`ret` to store return addresses on the stack, MARDU instead stores all return addresses in a shadow stack and leaves data destined for the regular stack untouched. Effectively, this protects all backward-edges. A MARDU `call` pushes a return trampoline address onto the shadow stack and jumps to a call trampoline; an instrumented `ret` directly jumps to the return trampoline address at the current top of the shadow stack. MARDU assumes a 64-bit address space and ability to leverage a segment register (*e.g.*, `%gs`); the base address of the MARDU shadow stack is randomized by ASLR and is hidden in `%gs`, which cannot be modified in userspace and will never be stored in memory.

Running example. Figure 3.2 is an example of executing a MARDU-compiled function `foo()`, which calls a function `bar()` and then returns. Every function call and return goes through trampoline code which stores the return address to a shadow stack. The body of `foo()` is entered via its call trampoline ❶. Before `foo()` calls `bar()`, the return trampoline address is stored onto the shadow stack. Control flow then jumps to `bar()`'s trampoline ❷, which will jump to the function body of `bar()` ❸. `bar()` returns to the address in the top of the shadow stack, which is

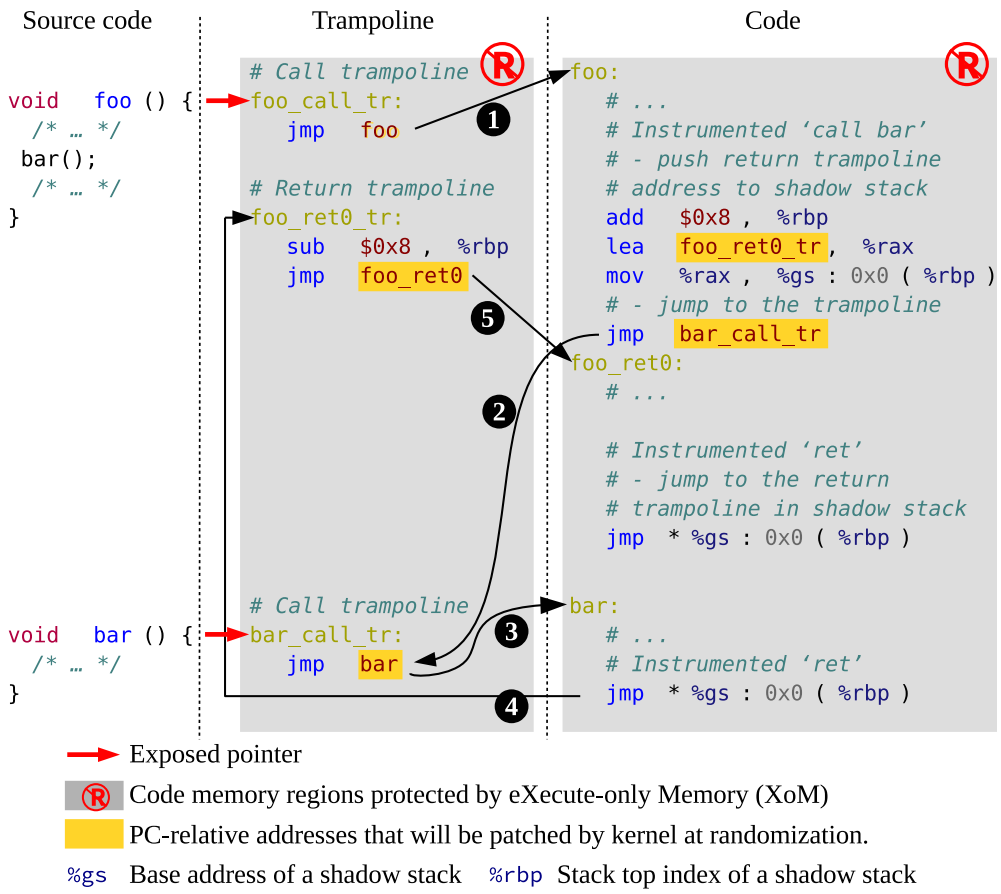


Figure 3.2: Illustrative example executing a MARDU-compiled function `foo()`, which calls a function `bar()` and then returns.

the return trampoline address ❹. Finally, the return trampoline returns to the instruction following the call in `foo()` ❺.

Enabling Code Sharing among Processes

PC-relative addressing. The MARDU compiler generates PC-relative (*i.e.*, position-independent) code so it can be shared amongst processes loading the same code in different virtual addresses. The key challenge here is *how to incorporate PC-relative addressing with randomization.* MARDU randomly places code (at function granularity) while trampoline regions remain stationary. This means any code using PC-relative addressing must be correspondingly patched once its randomized location is decided. In Figure 3.2, all jump targets between the trampoline and code, denoted in yellow rectangles, are PC-relative and must be adjusted. All data addressing instructions (*e.g.*, accessing global data, GOT, *etc.*) must also be adjusted.

Fixup information for patching. With this policy, it is necessary to keep track of these instructions to patch them properly during runtime. Similar to Compiler-assisted Code Randomization (CCR) [124], MARDU makes its runtime patching process simple and efficient by leveraging the LLVM compiler to collect and generate metadata, like fixups and relocations, into the binary describing exact locations for patching and their file-relative offset. Reading this information from the newly generated section in the executable, this fixup information makes patching as simple as just adjusting PC-relative offsets for given locations (see Figure 3.3). However, CCR only uses that information once, relying on a binary rewriter for a single static user-side binary executable randomization at function and basic-block granularity. Contrary to CCR, MARDU leverages the metadata added to allow processes to behave as runtime code rewriters, and re-randomize on-demand. The overhead of runtime patching is negligible because MARDU avoids "stopping the world" when patching the code to maintain internal consistency compared to other approaches, putting the burden on the crashed process. We elaborate on the patching process in Section 3.4.3.

Supporting a shared library. A call to a shared library is treated the same as a normal function call to preserve MARDU's code pointer hiding property; that is, MARDU refers to the call trampoline for the shared library call via procedure linkage table (PLT) or global offset table (GOT) whose address is resolved by the dynamic linker as usual. While MARDU does not specifically protect GOT, we assume that the GOT is already protected. For example, Fedora systems that support MPK have been hardened to enforce lazy binding will use a read-only GOT [71].

3.4.3 MARDU Kernel

The MARDU kernel randomizes code at load-time and runtime. It maps already-randomized code, if it exists, to the address space of a newly `fork`-ed process. When an application crashes, MARDU re-randomizes all mapped binaries associated with the crashing process and reclaims the previous randomized code from the cache after all processes are moved to a newly re-randomized code. MARDU prevents direct reading of randomized code from userspace using XoM. MARDU is also

responsible for initializing a shadow stack for each task[1].

Process Memory Layout

Figure 3.3 illustrates the memory layout of two MARDU processes. The MARDU compiler generates a PC-relative binary with trampoline code and fixup information ❶. When a binary is loaded to be mapped to a process with executable permissions, the MARDU kernel first performs a one time extraction of all MARDU metadata in the binary and associates it on a per-file basis. Extracting metadata gives MARDU the information it needs to perform (re-)randomization ❷. Note that load-time randomization and run-time re-randomization follow the exact same procedure. MARDU first generates a random offset to set apart the code and trampoline regions and then places functions in a random order within the code region. Once functions are placed, MARDU then uses the cached MARDU metadata to perform patching of offsets within both the trampoline and code regions to preserve program semantics. With the randomized code now semantically correct, it can be cached and mapped to multiple applications ❸.

Whenever a new task is created (`clone`), the MARDU kernel allocates a new shadow stack and copies the parent's shadow stack to its child; it is placed in the virtual code region created by the MARDU kernel. The base address of the MARDU shadow stack is randomized by ASLR and is hidden in segment register `%gs`. Any crash, such as brute-force guessing of base addresses, will trigger re-randomization, which invalidates all prior information gained, if any. To minimize the overhead incured from using a shadow stack, MARDU implements its own compact shadow stack without comparisons [32]. For our shadow stack implementation, we reserve one register, `%rbp`, to use exclusively as a stack top index of the shadow stack in order to avoid costly memory access.

Fine-Grain Code Randomization

Allocating a virtual code region. For each randomized binary, the MARDU kernel allocates a 2 GB *virtual* address region[2] (Figure 3.3 ❷), which will be mapped to userspace virtual address space with coarse-grained ASLR (Figure 3.3 ❸)[3]. The MARDU kernel positions the trampoline code at the end of the virtual address region and returns the start address of the trampoline via `mmap`. The trampoline address remains static throughout program execution even after re-randomization.

Randomizing the code within the virtual region. To achieve a high entropy, the MARDU kernel uses fine-grained randomization within the allocated virtual address region. Once the trampoline is positioned, the MARDU kernel randomly places non-trampoline code within the virtual address region; MARDU decides a *random offset* between the code and trampoline regions. Once

[1] In this paper, the term *task* denotes both process and thread as the convention in Linux kernel.

[2] We note that, for the unused region, we map all those virtual addresses to a single abort page that generates a crash when accessed to not to waste real physical memory and also detect potential attack attempts.

[3] We choose 2 GB because in x86-64 architecture PC-relative addressing can refer to a maximum of ±2 GB range from `%rip`.

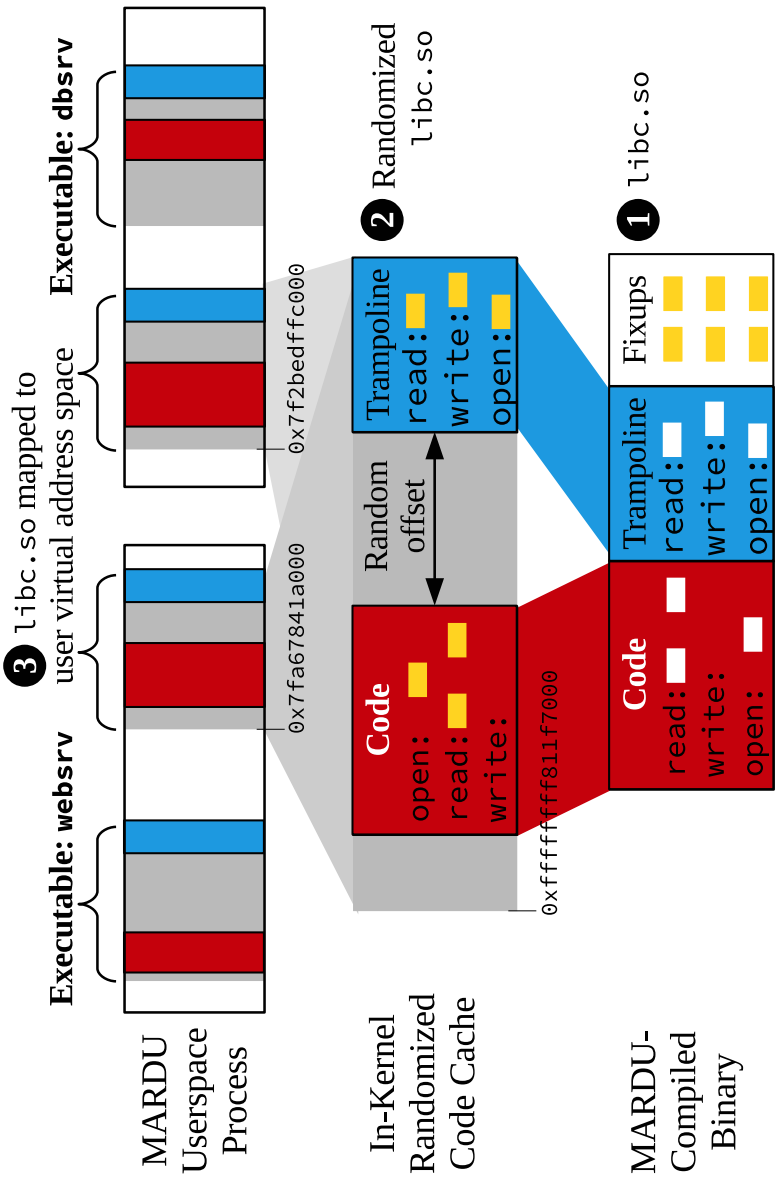


Figure 3.3: The memory layout of two MARDU processes: `websrv` (top left) and `dbsrv` (top right). The randomized code in kernel (`0xffffffff811f7000`) is shared by multiple processes, which is mapped to its own virtual base address (`0x7fa67841a000` for `websrv` and `0x7f2bedffc000` for `dbsrv`).

the code region is decided, MARDU permutes the function order within the code region to further increase entropy. As a result, trampoline addresses do not leak information on non-trampoline code and an adversary cannot infer any actual codes' location from the system information (*e.g.*, `/proc/<pid>/maps`) as they will get the same mapping information for the entire 2 GB region.

Patching the randomized code. After permuting functions, the MARDU kernel patches PC-relative instructions accessing code or data according to the randomization pattern. This patching process is trivial at runtime; the MARDU compiler generates fixup location information and the MARDU kernel re-calculates and patches PC-relative offsets of instructions according to the randomized function location. Note that patching includes control flow transfer between trampoline and non-trampoline code, global data access (*i.e.*, `.data`, `.bss`), and function calls to other shared libraries (*i.e.*, PLT/GOT).

Randomized Code Cache

The MARDU kernel manages a cache of randomized code. When a userspace process tries to map a file with executable permissions, the MARDU kernel first looks up if there already exists a randomized code of the file. If cache hits, the MARDU kernel maps the randomized code region to the virtual address of the requested process. Upon cache miss, it performs load-time randomization as described earlier. The MARDU kernel tracks how many times the randomized code region is mapped to userspace. If the reference counter is zero or system memory pressure is high, the MARDU kernel evicts the randomized code. Thus, in normal cases without re-randomization, MARDU randomizes a binary file only once (load-time). In MARDU, the randomized code cache is associated with the `inode` cache. Consequently, when the `inode` is evicted from the cache under severe memory pressure, its associated randomized code is also evicted.

Execute-Only Memory (XoM)

We designed XoM based on Intel MPK [97][4]. With MPK, each page is assigned to one of 16 domains under a *protection key*, which is encoded in a page table entry. Read and write permissions of each domain can be independently controlled through an MPK register. When randomized code is mapped to userspace, the MARDU kernel configures the XoM domain to be non-accessible (*i.e.*, neither readable nor writable in userspace), and assigns code memory pages to the created XoM domain, enforcing execute-only permissions. If an adversary tries to read XoM-protected code memory, re-randomization is triggered via the raised XoM violation. Unlike EPT-based XoM designs [207] that require system resources to enable virtualization as well as have inherent overhead from nested address translation, our MPK-based design does not impose such runtime overhead.

[4]As of this writing, Intel Xeon Scalable Processors [98] and Amazon EC2 C5 instance [9] support MPK. Other than x86, ARM AArch64 architecture also supports execute-only memory [11].

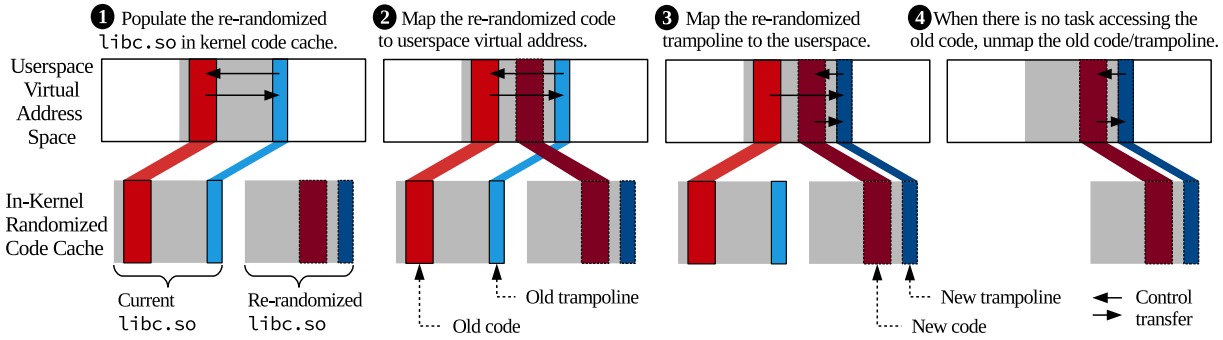


Figure 3.4: Re-randomization procedure in MARDU. Once a new re-randomized code is populated ❶, the MARDU kernel maps new code and trampoline in order ❷, ❸. This makes threads crossing the new trampoline migrate to the newly re-randomized code. After it is guaranteed that all threads are migrated to the new code, MARDU reclaims the old code ❹. Unlike previous continuous per-process re-randomization approaches, our re-randomization is time-bound, efficient, and system-wide.

On-Demand Re-randomization

Triggering re-randomization. When a process crashes, MARDU triggers re-randomization of *all* binaries mapped to the crashing process. Since MARDU re-randomization thwarts attacker's knowledge (*i.e.*, each attempt is an independent trial), an adversary must succeed in her first try without crashing, which is practically infeasible.

Re-randomizing code. Upon re-randomization, the MARDU kernel first populates another copy of the code (*e.g.*, `libc.so`) in the code cache and freshly randomizes it (Figure 3.4 ❶). MARDU leaves trampoline code at the same location to avoid mutating code pointers but it does randomly place non-trampoline code (via new random offset) such that the new version does not overlap with the old one. Then, it permutes functions in the code. Thus, re-randomized code is completely different from the previous one without changing trampoline addresses.

Live thread migration without stopping the world. Re-randomized code prepared in the previous step is not visible to userspace processes because it is not yet mapped to userspace. To make it visible, MARDU first maps the new non-trampoline code to the application's virtual address space, Figure 3.4 ❷. The old trampolines are left mapped, making new code not reachable. Once MARDU remaps the virtual address range of the trampolines to the new trampoline code by updating corresponding page table entries ❸, the new trampoline code will transfer control flow to the new non-trampoline code. Hereafter any thread crossing the trampoline migrates to the new non-trampoline code without stopping the world.

Safely reclaiming the old code. MARDU can safely reclaim the code only after all threads migrate to the new code ❹. MARDU uses *reference counting* for each randomized code to check if there is a thread accessing the old code. After the new trampoline code is mapped ❸, MARDU sets

a reference counter of the old code to the number of all *runnable* tasks [5] that map the old code. It is not necessary to wait for the migration of a non-runnable, sleeping task because it will correctly migrate to the newest randomized code region when it passes through the return trampoline, which refers to the new layout when it wakes up. The reference counter is decremented when a runnable task enters into the MARDU kernel due to system call or preemption. When calling a system call, the MARDU kernel will decrement reference counters of all code that needs to be reclaimed. When the task returns to userspace, it will return to the return trampoline and the return trampoline will transfer to the new code. When a task is preempted out, it may be in the middle of executing the old non-trampoline code. Thus, the MARDU kernel not only decrements reference counters but also translates `%rip` of the task to the corresponding address in the new code. Since MARDU permutes at function granularity, `%rip` translation is merely adding an offset between the old and new function locations.

Summary. Our re-randomization scheme has three nice properties: 1) time boundness of re-randomization, 2) almost zero overhead of running process, and 3) system-wide re-randomization. Because MARDU migrates *runnable* tasks at system call and scheduling boundaries, it ensures that MARDU re-randomization will always guarantee the process to use the newly secure version of MARDU-enabled code once awoken or has crossed the system call boundary. Just as important, processes will *never* have access to the attacker exposed code ever again once crossing those boundaries. If another process crashes in the middle of re-randomization, MARDU will not trigger another re-randomization until the current randomization finishes. However, as soon as the new randomized code is populated ❶, a new process will map the new code immediately. Therefore, the old code cannot be observed more than once. The MARDU kernel populates a new randomized code in the context of a crashing process. All other runnable tasks only additionally perform reference counting or translation of `%rip` to the new code. Thus, its runtime overhead for runnable tasks is negligible. *To the best of our knowledge,* MARDU *is the first system to perform system-wide re-randomization allowing code sharing.*

3.5 Implementation

We implemented MARDU on the Linux x86-64 platform. The MARDU compiler is implemented using LLVM 6.0.0 and the MARDU kernel is implemented based on Linux kernel 4.17.0 modifying 3549 and 4009 lines of code (LoC), respectively. We used `musl libc` 1.1.20 [72], a fast, lightweight C standard library for Linux. We chose `musl libc` because `glibc` is not able to be compiled with LLVM/Clang. We manually wrapped all inline assembly functions present in `musl` to allow them to be properly identified and instrumented by the MARDU compiler. We modified 164 LoC in `musl libc` for the wrappers.

[5] A task in a `TASK_RUNNING` status in Linux kernel.

3.5.1 MARDU Compiler

Trampoline. The MARDU compiler is implemented as backend target-ISA (x86) specific `MachineFunctionPass`. This pass instruments each function body as described in § 3.4.2.

Re-randomizable code. The following compiler flags are used by the MARDU compiler: `-fPIC` enables instructions to use PC-relative addressing; `-fomit-frame-pointer` forces the compiler to relinquish use of register `%rbp`, as register `%rbp` is repurposed as the stack top index of a shadow stack in MARDU; `-mrelax-all` forces the compiler to always emit full 4-byte displacement in the executable, such that the MARDU kernel can use the full span of memory within our declared 2GB virtual address region and maximize entropy when performing patching; lastly, the MARDU compiler ensures code and data are segregated in different pages via using `-fno-jump-tables` to prevent false positive XoM violations.

3.5.2 MARDU Kernel

Random number generation. MARDU uses a cryptographically secure random number generator in Linux based on hardware instructions (*i.e.*, `rdrand`) in modern Intel architectures. Alternatively, MARDU can use other secure random sources such as `/dev/random` or `get_random_bytes()`.

3.5.3 Limitation of Our Prototype Implementation

Assembly Code. MARDU does not support inline assembly as in `musl`; however, this could be resolved with further engineering. Our prototype uses wrapper functions to make assembly comply with MARDU calling convention.

Setjmp and exception handling. MARDU uses a shadow stack to store return addresses. Thus, functions such as `setjmp`, `longjmp`, and `libunwind` that directly manipulate return addresses on stack are not supported by our prototype. Adding support for these functions could be resolved by porting these functions to understand our shadow stacks semantics, as our shadow stack is a variant of compact, register-based shadow stack [32].

C++ support. Our prototype does not support C++ applications since we do not have a stable standard C++ library that is `musl`-compatible.

3.6 Evaluation

We evaluate MARDU by answering these questions:

- How secure is MARDU, when presented against current known attacks on randomization? (§ 3.6.1)

- How much performance overhead does the needed instrumentation of MARDU impose, particularly for compute-intensive benchmarks in a typical runtime without any attacks? (§ 3.6.2)
- How scalable is MARDU in terms of load time, re-randomization time with and without on-going attacks, and memory savings, paricularly for concurrent processes such as in a real-world network facing server? (§ 3.6.3)

Applications. We evaluate the performance overhead of MARDU using SPEC CPU2006. The SPEC CPU benchmark suite has various realistic compute-intensive applications, (*e.g.*, `gcc`) ideal to see worst-case performance overhead of MARDU. We tested 12 C language benchmarks using input size *ref*; C++ benchmarks are excluded as our current prototype does not support C++. We chose to use SPEC CPU2006 over the more recent SPEC CPU2017 benchmark suite to easily compare MARDU to prior relevant re-randomization techniques. We test performance and scalability of MARDU on a complex, real-world multi-process web server with NGINX.

Experimental setup. Our experiments are performed on a machine housing two Intel Xeon Silver 4116 CPUs (2.10 GHz) which feature 24-cores (48-hardware threads) and 128 GB DRAM. All programs are compiled with CFLAGS `-O2`.

3.6.1 Security Evaluation

We analyze the resiliency of MARDU against existing attacker models with load-time randomization (A1–A2, Section 3.6.1) and continuous re-randomization. (A3–A4, Section 3.6.1). Then, to illustrate the effectiveness of MARDU for a wider class of code-reuse attacks beyond ROP, we discuss the threat model of NEWTON [222] with MARDU (Section 3.6.1).

Attacks against Load-Time Randomization

Against JIT-ROP attacks (A1). MARDU asserts permissions for all code areas and trampoline regions as execute-only (via XoM); thereby, JIT-ROP cannot read code contents directly.

Against code inference attacks (A2). MARDU blocks code inference attacks, including BROP [25], clone-probing [141], and destructive code read attacks [173, 200] via layout re-randomization triggered by an application crash or XoM violation. Every re-randomization renders all previously gathered (if any) information regarding code layout invalid and therefore prevents attackers from accumulating indirect information. Note that attacks such as Address-Oblivious Code Reuse (AOCR) [185], do not fall into the category of A2. This attack vector's process of control hijacking more closely resembles full-function code re-use rather than indirect exposure of code; AOCR leverages manipulation of data and function pointer corruption and does not require usage of `ret` gadgets.

Hiding shadow stack. Attackers with arbitrary read/write capability (A1/A2) may attempt to leak/alter shadow stack contents if its address is known. Although the location of the shadow

stack is hidden behind the `%gs` register, attackers may employ attacks that undermine this sparse-memory based information hiding [67, 85, 165]. To prevent such attacks, MARDU reserves a 2 GB virtual memory space for the shadow stack (the same way MARDU allocates code/library space) and chooses a random offset to map the shadow stack; all other pages in the 2 GB space are mapped as an abort page. Regarding randomization entropy of shadow stack hiding, we take an example of a process that uses sixteen pages for the stack. In such a case, the possible shadow stack positions are:

$$\begin{aligned} \#\ of\ positions &= (MEMSIZE - STACKSIZE)/PAGESIZE \\ &= (2^{31} - 16 * 2^{12})/2^{12} = 524,272 \end{aligned} \tag{3.1}$$

thereby, the probability of successfully guessing a valid shadow stack address is one in 524,272, practically infeasible. Even assuming if attackers are able to identify the 2 GB region for the shadow stack, they must also overcome the randomization entropy of the offset to get a valid address within this region (winning chance: roughly one in 2^{31}, as MARDU's shadow stack can start at an arbitrary address within a page and not align to the 4K-page boundary); any incorrect probe will generate a crash, trigger re-randomization, thwarting the attack.

Entropy. MARDU applies both function-level permutation and random start offset to provide a high entropy to the randomized code layout. In particular, MARDU permutates all functions in each executable at each time of randomization. In this way, randomization entropy (E_{func}) depends on the number of functions in the executable (n), and the entropy gain can be formulated as:

$$E_{func} = log_2(n!) \tag{3.2}$$

Additionally, MARDU applies a random start offset to the code area in 2 GB space in each randomization. Because the random offset could be anywhere in 2 GB range excluding the size of trampoline region and twice the size of the program (to avoid overlapping), the entropy gain by the random offset (E_{off}) can be formulated as:

$$E_{off} = log_2(2^{31} - sizeof(trampoline) - 2{\times}sizeof(program)) \tag{3.3}$$

and the total entropy that MARDU provides is:

$$E_{\text{MARDU}} = E_{func} + E_{off} \tag{3.4}$$

We take an example of `470.lbm` in SPEC CPU2006, a case which provides the minimum entropy in our evaluation. The program contains 16 functions, and the entire size of the program including trampoline instrumentation is less than 64 KB. In such a case, the total entropy is:

$$\begin{aligned} E_{func} &= log_2(16!) > 44.25, \\ E_{off} &= log_2(2^{31} - 2 \times 64K) > 30.99 \end{aligned} \tag{3.5}$$

$$E_{\text{MARDU}} = E_{func} + E_{off} > 74.24 \tag{3.6}$$

Therefore, even for a small program, MARDU randomizes the code with significantly high entropy (74 bits) to render an attacker's success rate for guessing the layout negligible.

Attacks against Continuous Re-randomization

Against low-profile attacks (A3). MARDU does not rely on timing nor system call history for triggering re-randomization. As a result, neither low-latency attacks nor attacks without involving system calls are effective against MARDU. Instead, re-randomization is triggered and performed by any MARDU instrumented application process that encounters a crash (*e.g.*, XoM violation). Nonetheless, a potential A3 vector could be one that does not cause any crash during exploitation (*e.g.*, attackers may employ crash-resistant probing [67, 75, 85, 122, 165]). In this regard, MARDU places all code in execute-only memory within 2 GB mapped region. Such a stealth attack could only identify multiples of 2 GB code regions and will fail to leak any layout information.

Against code pointer offsetting attacks (A4). Attackers may attempt to launch this attack by adding/subtracting offsets to a pointer. To defend against this, MARDU decouples any correlation between trampoline function *entry* addresses and function *body* addresses (*i.e.*, no fixed offset), so attackers cannot refer to the middle of a function for a ROP gadget without actually obtaining a valid function body address. Additionally, the trampoline region is also protected with XoM, thus attackers cannot probe it to obtain function body addresses to launch A4. MARDU limits available code-reuse targets to only exported functions in the trampoline.

Beyond ROP attacks

Attack analysis with NEWTON. To measure the boundary of viable attacks against MARDU, we present a security analysis of MARDU based on the threat model set by NEWTON [222]. In this regard, we analyze possible writable pointers that can change the control flow of a program (write constraints) as well as possible available gadgets in MARDU (target constraints), which will reveal what attackers can do under this threat model. In short, MARDU allows only the reuse of exported functions via call trampolines.

For write constraints, attackers cannot overwrite real code addresses such as return addresses or code addresses in the trampoline. MARDU only allows attackers to overwrite other types of pointer memory, *e.g.*, object pointers and pointers to the call trampoline. For target constraints, attackers

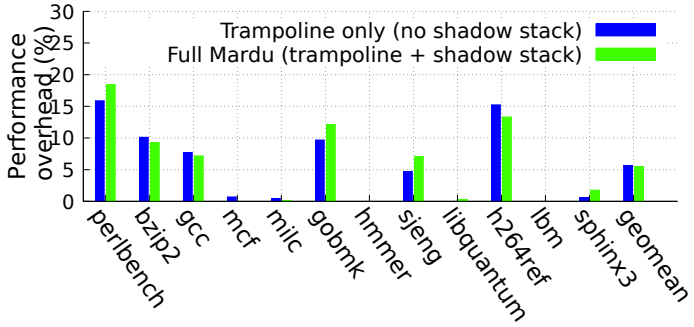


Figure 3.5: MARDU performance overhead breakdown for SPEC

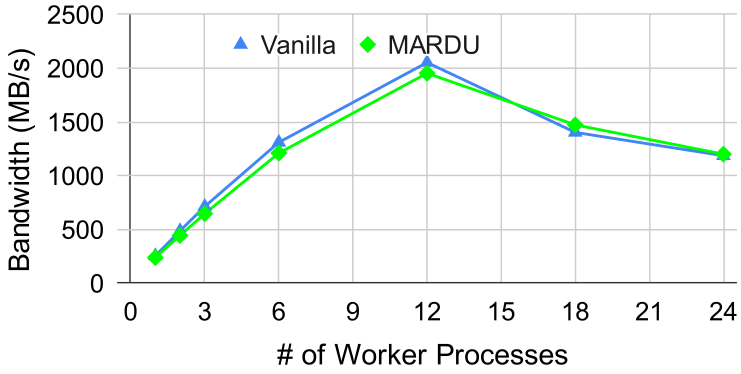


Figure 3.6: Performance comparison of NGINX web server

can reuse only the exported functions via call trampoline. Note that a function pointer is a reusable target in any re-randomization techniques using immutable code pointers [41, 224, 232]. Although MARDU allows attackers to reuse function pointers in accessible memory (*e.g.*, a function pointer in a structure), such live addresses will never include real code addresses or return addresses, and will be limited to addresses only referencing call trampolines. *Under these write and target constraints, inferring the location of ROP gadgets from code pointers (*e.g.*, leaking code addresses or adding an offset) is not possible.*

3.6.2 Performance Evaluation

Runtime performance overhead with SPEC CPU2006. Figure 3.5 shows the performance overhead of SPEC with MARDU trampoline only instrumentation (which does not use a shadow stack) as well as with a full MARDU implementation. Both of these numbers are normalized to the unprotected and uninstrumented baseline, compiled with vanilla Clang. Note that this performance overhead is the base incured overhead of security hardening an application with MARDU. In the rare case, that the application were to come under attack, on-demand re-randomization would be triggered inducing additional brief performance overheads. We discuss the performance overhead

of MARDU under active attack in § 3.6.3.

Figure 3.5 does not include a direct performance comparison to other randomization techniques as MARDU is substantially different in how it implements re-randomization and the source code of closely related systems, such as Shuffler [232] and CodeArmor [41], is not publicly available. It is not based on timing nor system call history compared to previous works. This peculiar approach allows MARDU's average overhead to be comparable to the fastest re-randomization systems and its worst-case overhead significantly better than similar systems. The average overhead of MARDU is 5.5%, and the worst-case overhead is 18.3% (`perlbench`); in comparison to Shuffler [232] and CodeArmor [41], whose reported average overheads are 14.9% and 3.2%, and their worst-case overhead are 45% and 55%, respectively (see Table 3.1). TASR [22] shows a very practical average overhead of 2.1%; however, it has been reported by Shuffler [232] and ReRanz [224] that TASR's overhead against a more realistic baseline (not using compiler flag `-Og`) is closer to 30-50% overhead. This confirms MARDU is capable of matching if not slightly improving the performance (especially worst-case) overhead, while casting a wider net in terms of known attack coverage.

MARDU's two sources of runtime overhead are trampolines and the shadow stack. MARDU uses a compact shadow stack without a comparison epilogue whose sole purpose is to secure return addresses. Specifically, only 4 additional assembly instructions are needed to support our shadow stack. Therefore we show the trampoline only configuration to clearly differentiate the overhead contribution of each component. Figure 3.5 shows MARDU's shadow stack overhead is negligible with an average of less than 0.3%, and in the noticeable gaps, adding less than 2% in `perlbench`, `gobmk`, and `sjeng`. The overhead in these three benchmarks comes from the higher frequency of short function calls, making shadow stack updates not amortize as well as in other benchmarks. In the cases where Full MARDU is actually faster than the Trampoline only version (*e.g.*, `bzip2`, `gcc`, and `h264ref`), we investigated and found that our handcrafted assembly for integrating the trampolines with the regular stack in the Trampoline only version can inadvertantly cause elevated amounts of branch-misses, leading to the expected performance slowdown.

3.6.3 Scalability Evaluation

Runtime performance overhead with NGINX. NGINX is configured to handle a max of 1024 connections per processor, and its performance is observed according to the number of worker processes. `wrk` [83] is used to generate HTTP requests for benchmarking. `wrk` spawns the same number of threads as NGINX workers and each `wrk` thread sends a request for a 6745-byte static html. *To see worst-case performance, `wrk` is run on the same machine as NGINX to factor out network latency unlike Shuffler.* Figure 3.6 presents the performance of NGINX with and without MARDU for a varying number of worker processes. The performance observed shows that MARDU exhibits very similar throughput to vanilla. MARDU incurs 4.4%, 4.8%, and 1.2% throughput degradation on average, at peak (12 threads), and at saturation (24 threads), respectively. Note that Shuffler [232] suffers from overhead from its *per-process* shuffling thread; just enabling Shuffler essentially doubles CPU usage. *Even in their NGINX experiments with network latency (*i.e.*, running a benchmarking client on a different machine), Shuffler shows 15-55% slowdown.* This verifies MARDU's design that having a crashing process perform system-wide re-randomization, rather than a per-process background thread as in Shuffler, scales better.

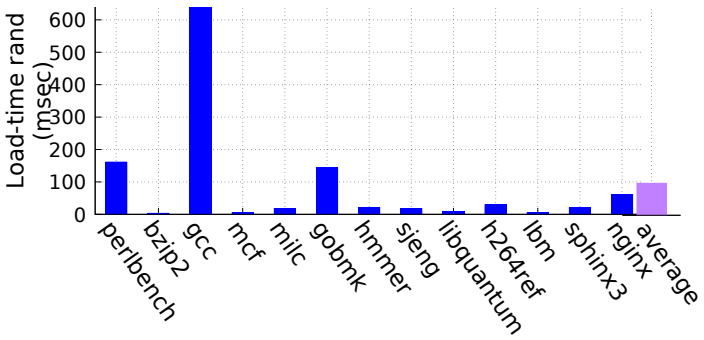


Figure 3.7: Cold load-time randomization overhead

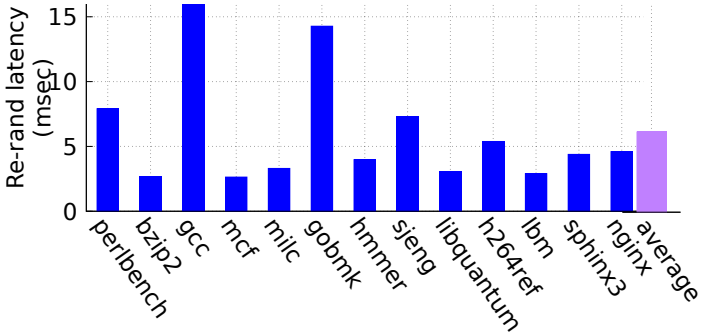
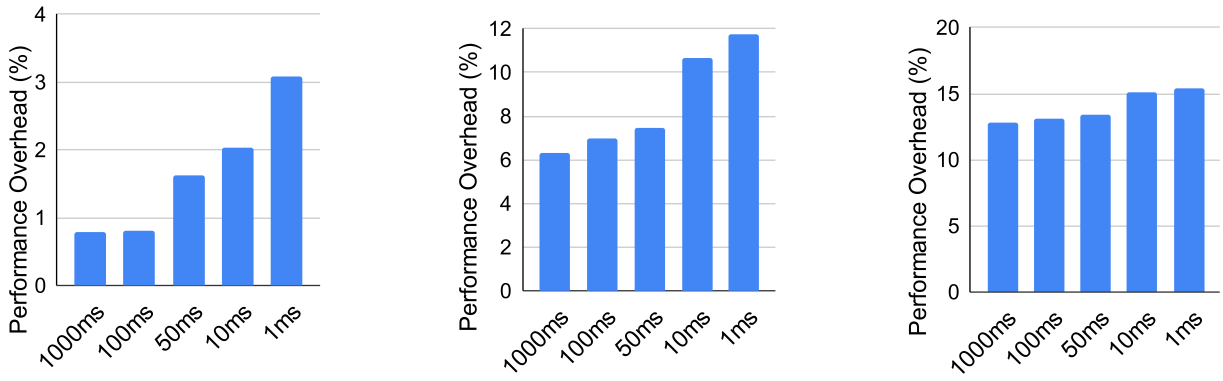


Figure 3.8: Runtime re-randomization latency

Load-time randomization overhead. We categorize load-time to cold or warm load-time whether the in-kernel code cache (❷ in Figure 3.3) hits or not. Upon a code cache miss (*i.e.*, the executable is first loaded in a system), MARDU performs initial randomization including function-level permutation, start offset randomization of the code layout, and loading & patching of fixup metadata. As Figure 3.7 shows, all C SPEC benchmarks showed negligible overhead averaging 95.9 msec. `gcc`, being the worst-case, takes 771 msec; it requires the most (291,699 total) fixups relative to other SPEC benchmarks, with ≈9,372 fixups on average. `perlbench` and `gobmk` are the only other outliers, having 103,200 and 66,900 fixups, respectively; all other programs have <<35K fixups (refer to Table 3.2). For NGINX, we observe that load time is constant (61 msec) for any number of specified worker processes. Cold load-time is roughly linear to the number of trampolines. Upon a code cache hit, MARDU simply maps the already-randomized code to a user-process's virtual address space. Therefore we found that warm load-time is negligible. Note that, for a cold load-time of `musl` takes about 52 msec on average. Even so, this is a one time cost; all subsequent warm load-time accesses of fetching `musl` takes below 1μsec, for any program needing it. Thus, load time can be largely ignored.

Re-randomization latency. Figure 3.8 presents time to re-randomize all associated binaries of a crashing process. The time includes creating & re-randomizing a new code layout, and re-claiming old code (❶-❹ in Figure 3.4). We emulate an XoM violation by killing the process via a `SIGBUS` signal and measured re-randomization time inside the kernel. The average latency of SPEC is 6.2 msec. The performance gained between load-time and re-randomization latency is



(a) milc **(b)** sjeng **(c)** gobmk

Figure 3.9: Overhead varying re-randomization frequency

from MARDU taking advantage of metadata being cached from load-time, meaning no redundant file I/O penalty is incurred. To evaluate the efficiency of re-randomization on multi-process applications, we measured the re-randomization latency with varying number of NGINX worker processes up to 24. We confirm latency is consistent regardless of number of workers (5.8 msec on average, 0.5 msec std. deviation).

Re-randomization overhead under active attacks. In addition, a good re-randomization system should exhibit good performance not only in its idle state but also under stress from active attacks. To evaluate this, we stress test MARDU under frequent re-randomization to see how well it can perform, assuming a scenario that MARDU is under attack. In particular, we measure the performance of SPEC benchmarks while triggering frequent re-randomization. We emulate the attack by running a background application, which continuously crashes at the given periods: 1 sec, 100 msec, 50 msec, 10 msec, and 1 msec. SPEC benchmarks and the crashing application are linked with the MARDU version of `musl`, forcing MARDU to constantly re-randomize `musl` and potentially incur performance degradation on other processes using the same shared library. In this experiment, we choose three representative benchmarks, `milc`, `sjeng`, and `gobmk`, that MARDU exhibits a small, medium, and large overhead in an idle state, respectively. Figure 3.9 shows that the overhead is consistent, and in fact, is very close to the performance overhead in the idle state observed in Figure 3.5. More specifically, all three benchmarks differ by less than 0.4% at a 1 sec re-randomization interval. When we decrease the re-randomization period to 10 msec and 1 msec, the overhead is quickly saturated. Even at 1 msec re-randomization frequency, the additional overhead is under 6 %. These results show that MARDU provides performant system-wide re-randomization even under active attack.

File size overhead. Figure 3.10 and Table 3.2 show how much binary files increase with MARDU compilation. In our implementation, file size increase comes from transforming the traditional x86-64 calling convention with the one designed for MARDU (besides calls to outside libraries). On average, MARDU compilation with trampolines increases the file size by 66%. One would assume that applications with more call sites incur a higher overhead as we are adding 5 instructions for every `call` and 4 instructions for every `retq` (*e.g.*, `perlbench`, `gcc`, `milc`, & `gobmk` are the only

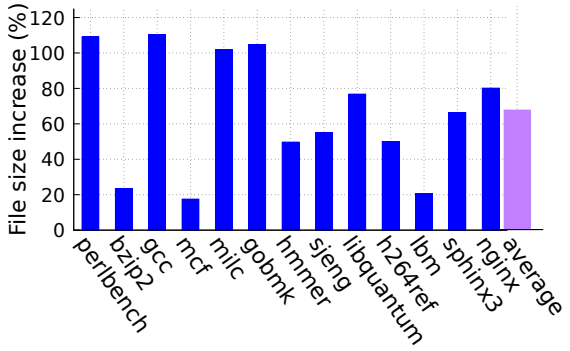


Figure 3.10: File size increase with MARDU compilation

Table 3.2: Breakdown of MARDU instrumentation

Benchmark	Numbers of Fixups				Binary Increase (bytes)		
	Call Tr.	Ret Tr.	PC-rel. addr	Total	Trampolines	Metadata	Total
perlbench	1596	39174	62430	103200	1115136	2607559	3722695
bzip2	66	926	896	1888	17568	78727	96295
gcc	4015	118617	169067	291699	3074672	6276870	9351542
mcf	23	94	208	325	1824	19056	20880
milc	234	3531	7256	11021	110688	313620	424308
gobmk	2388	22880	41632	66900	726176	3085208	3811384
hmmer	452	5145	9925	15522	139216	574446	713662
sjeng	129	1368	5418	6915	58912	250234	309146
libquantum	97	1659	1424	3180	25952	93222	119174
h264ref	508	5874	14824	21206	278240	714629	992869
lbm	16	75	260	351	1920	16549	18469
sphinx3	308	4958	8010	13276	103920	409814	513734
NGINX	1497	15004	18984	35485	416736	1309708	1726444
musl libc	4400	10009	7594	22003	192153	1238071	1430224

benchmarks with over 100% increase, being 108%, 110%, 101%, & 104% respectively).

Runtime memory savings. While there is an upfront one-time cost for instrumenting with MARDU, the savings greatly outweigh this. To illustrate, we show a typical use case of MARDU in regards to shared code. `musl` is ≈800 KB in size, instrumented is 2 MB. Specifically, `musl` has 14K trampolines and 7.6K fixups for PC-relative addressing, the total trampoline size is 190 KB and the amount of loaded metadata is 1.2 MB (Table 3.2). Since MARDU supports code sharing, only one copy of `libc` is needed for the entire system. Backes *et al.* [13] and Ward *et al.* [227] also highlighted the code sharing problem in randomization techniques and reported a similar amount of memory savings by sharing randomized code. Finally, note that the use of our shadow stack does not increase the runtime memory footprint beyond the necessary additional memory page allocated to support the shadow stack and the increase in code size from our shadow stack instrumentation. MARDU solely relocates return addresses from the normal stack to the shadow stack.

System-wide Performance Estimation. Deploying MARDU system-wide for all applications and all shared libraries requires additional engineering effort of recompiling the entire Linux distribution. Instead, we get an estimate of how MARDU would perform on a regular Linux server during boot time. We obtain this estimate based on the fact that MARDU's load-time overhead increases linearly with the total number of functions and call sites present in an application or library. We calculate the estimated boot overhead if the entire system was protected by MARDU. Referencing Figure 3.2, MARDU requires one trampoline per function containing one fixup, and one return trampoline per callsite containing three fixups. In addition, all PC-relative instructions must be patched. Therefore the total number of fixups to be patched is as follows:

$$Total\ \#\ Fixups = \#\ Functions + (\#\ Callsites * 3) + \#\ PC\ relative\ Instructions \tag{3.7}$$

Extrapolating from MARDU 's load-time randomization overhead in Figure 3.7, where `gcc` has most fixups at 291,699 and takes 771 ms, this makes each fixup take approximately 2.6 μsec. We recorded all executables launched as well as all respective loaded libraries in our Linux server to calculate the additional overhead imposed by MARDU during boot time. We included all programs run within the first 5 minutes of boot time as well as looking at the current system load. In five minutes after the system booting, we recorded a total of 117 no longer active processes and recorded 265 currently active processes, using a total of 784 unique libraries. The applications contained a total of 8,862 functions, a total of 472,530 callsites, and 415,951 total PC-relative instructions. Using Equation 3.7 from above, this gave a total of 1,842,403 fixups if all launched applications were MARDU enabled. The libraries contained a total of 223,415 functions, a total of 4,450,488 callsites, and 2,514,676 total PC-relative instructions; this gave a total of 16,089,555 fixups for shared libraries. Using our estimation from `gcc`, we can approximate that patching all fixups including both application fixups and shared library fixups (a total of 17,931,958 fixups) for a MARDU enabled Linux server will take roughly ≈46.6 additional seconds, compared to a vanilla boot. To give a little more insight, application fixups contribute only ≈4.8 seconds of delay; the majority of overhead comes from randomization of the shared libraries. However, note that this delay is greatly amortized as many libraries are shared by a large number of applications, compared to the scenario where each library is not shared (*e.g.*, statically-linked) and needs a separate randomized copy for each application requiring it.

System-wide Memory Savings Estimation. Similarly, we give a system-wide snapshot of memory savings observed when MARDU's randomized code sharing is leveraged. For this, we again use the same Linux server for system-wide estimation. The vanilla total file size of 784 unique libraries is approximately 787 MB. From our scalability evaluation, Figure 3.10, showing that MARDU roughly increases file size by 66% on average, this total file size would grow to 1,306 MB if all were instrumented with MARDU. While this does appear to be a large increase, it is a one time cost as code sharing is enabled under MARDU. From our Linux server having 265 processes, 127 of had mapped libraries. If code sharing is not supported, each process needs its own copy of a library in memory. We counted each library use and multiplied by its size to get the total non-sharing memory usage. For our Linux server, this non-sharing would incur approximately a 8.8 GB overhead. Meaning, MARDU provides approximately 7.5 GB memory savings through its inherent

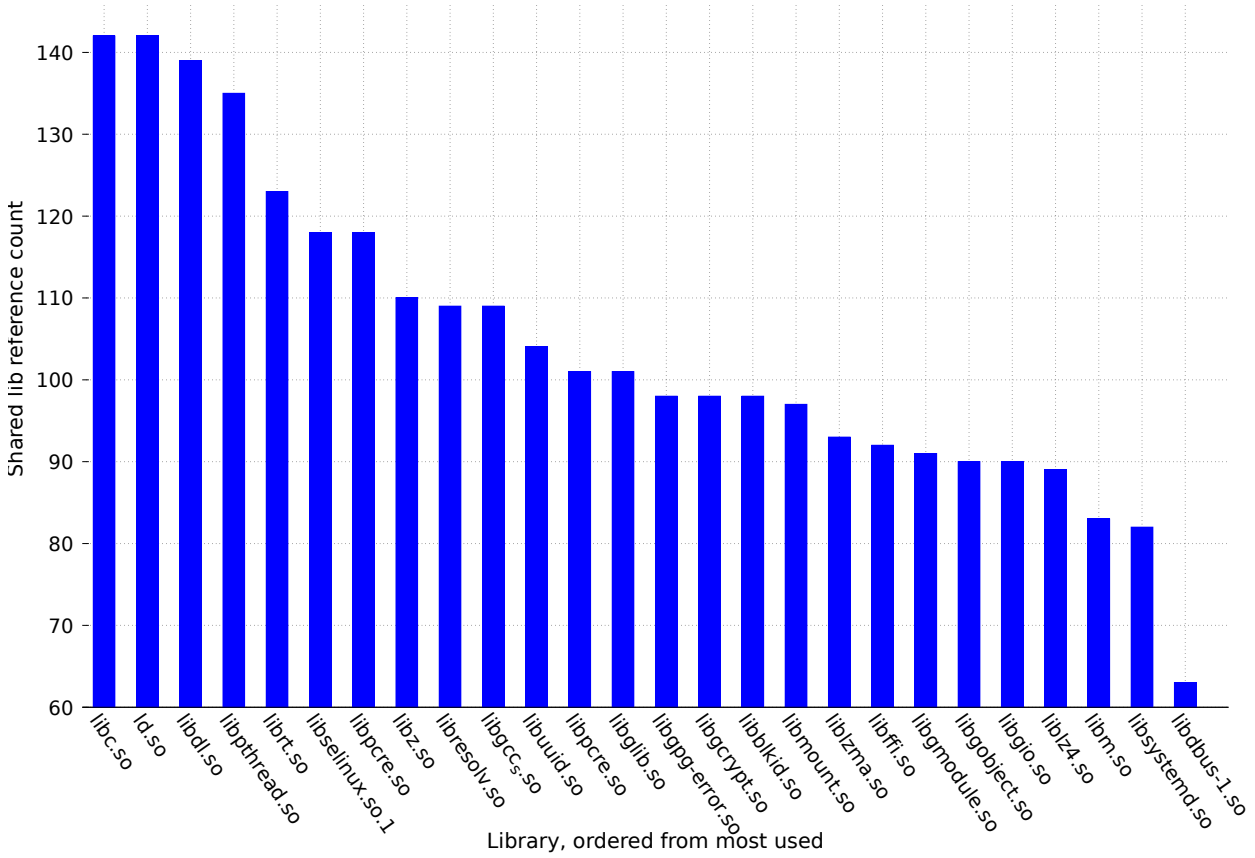


Figure 3.11: Top 25 shared libraries with their reference count on our idle Linux server ordered from most linked to least linked libraries.

code sharing design. This memory savings is compared to if these libraries were individually and separately statically linked to each of the running processes.

To get how many times each library is shared by multiple processes, we analyzed each process's memory mapping on our Linux server by investigating `/proc/{PID}/maps`. Figure 3.11 presents the active reference count for the 25 most linked shared libraries on our idle Linux server. The 25 most linked shared libraries are referenced over ≈106 times on average, showing that dynamically linked libraries really do save a lot of memory compared to a non-shared approach. For the same 25 most linked shared libraries, we also demonstrate in Figure 3.12 the estimated memory savings obtained for each of those libraries if MARDU was used instead of an approach that does not support sharing of code. Notice that some of our biggest memory savings come from `libc.so` and `libm.so`, very commonly used libraries that MARDU saves almost 0.80 GB and 0.25 GB of memory for, respectively.

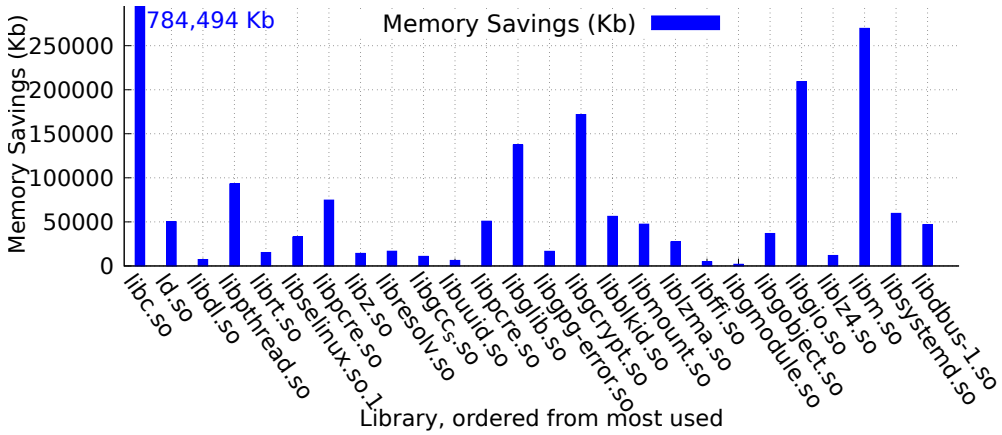


Figure 3.12: Estimated runtime memory savings with shared memory MARDU approach for the top 25 most linked libraries on our idle Linux server.

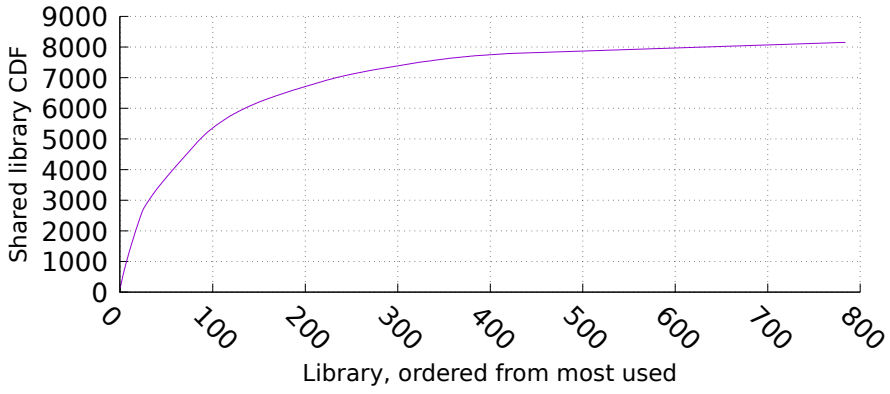


Figure 3.13: CDF of shared library occurance on a idle Linux server.

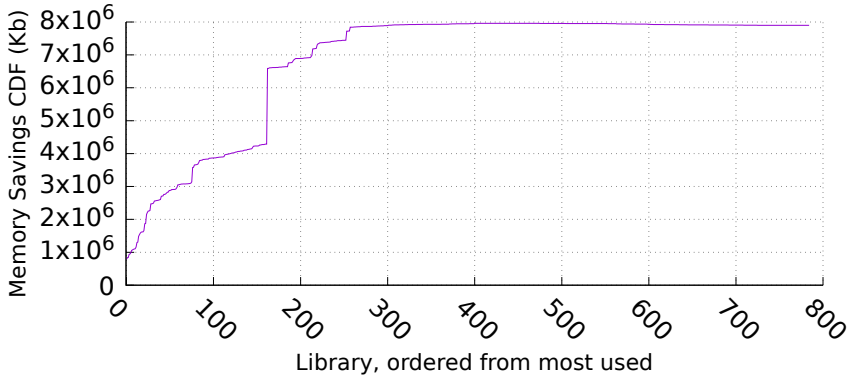


Figure 3.14: CDF plot of estimated runtime memory savings with MARDU's shared memory approach.

We also show the entire system snapshot (including all 784 unique libraries) in the form of a CDF for both the unique library link count in Figure 3.13 and cumulative memory savings in Figure 3.14 if MARDU were to be applied and used system wide for all dynamically linked libraries. From Figure 3.13, it can be seen that approximately 150 libraries are in the 75th percentile of link count.

3.7 Discussion and Limitations

Applying MARDU to binary programs. Although our current MARDU prototype requires access to source code, applying MARDU directly to binary programs is possible. MARDU requires detecting all function call transfers (`call`/`ret`) and instrumenting them to use trampolines in order to keep control transfers semantically correct with re-randomization. A potential way to enable this is to apply techniques that can retrieve precise disassembly from a given binary, such as BYTEWEIGHT [16], to identify possible call targets. A more recent innovation, Egalito [233], a binary recompiler, showed that it is possible to raise (stripped) modern Linux binaries into a low level intermediate representation (IR). Their standalone, layout-agnostic IR is precise and allows arbitrary modifications. This approach would allow binary code or legacy binaries to be directly instrumented such that transfers utilize trampolines via binary re-writing. Leveraging re-assembleable assembly, Retrowrite [61], also offers a binary transformation framework for 64-bit position-independent binaries. This work makes it plausible that MARDU could leverage their underlying binary-rewriting framework to instrument a popular and important class of binaries, as well as notably third-party shared libraries. Either of these recent approaches would enable MARDU to perform security hardening to software distributed as binaries to end-users.

Security of Protection Keys for Userspace. As briefly mentioned in Section 3.3, native MPK applications containing `wrpkru` instructions are not supported on a MARDU system. If there exists a running native MPK application that contains `wrpkru` instructions on the host system, there *does exist* an attack scenario that could victimize this native MPK application and break

the guarantees and assumptions set by MARDU. To elaborate, if any applications have `wrpkru` instructions, it possible for a cross-process MPK attack to occur such that all MARDU guarded code (protected under an XoM domain) could be made accessible to the attacker, (*i.e.*, this could done via an attack vector not covered by MARDU, such as data/argument corruption) leveraging the secondary process containing `wrpkru` instructions, and effectively disabling MARDU.

To remedy this current limitation, MARDU could be extended to include and use HODOR [89] or ERIM [217] style approaches. Instrumenting hardware watchpoints to vet `wrpkru` instruction execution at runtime, or perform binary rewriting to generate functionally equivalent assembly where unintended `wrpkru` instructions occur in the code region, respectively, would then allow MARDU to support and properly protect applications containing native `wrpkru` instructions.

Full-function reuse attacks. Throughout our analysis, we show that existing re-randomization techniques that use a function trampoline or indirection table [41, 232], *i.e.*, use immutable (indirect) code pointer across re-randomization, cannot prevent full-function reuse attacks. This also affects MARDU; although limited to functions exposed in the trampoline, MARDU cannot defend against an attacker re-using such exposed immutable code pointers as gadgets by leaking code pointers and believe that this is a limitation of using immutable code pointers.

That being said, a possible workaround could be to utilize a monitoring mechanism limited to tracking function pointer assignment. While this approach would be cumbersome, it will have a much smaller overhead because of its smaller scope leading to being more secure while producing less overhead than Shuffler [232].

Another possible solution to prevent these attacks could be pairing MARDU together with control-flow-integrity (CFI) [1, 42, 77, 86, 87, 139, 161, 163, 170, 178, 212, 219, 221, 235, 238], code-pointer integrity/separation (CPI/CPS) [126], or other hardware-assisted solutions such as Intel's Control-flow Enforcement Technology (CET) [96] and ARM's Pointer Authentication Code (PAC) [182]. MARDU's defense is orthogonal to forward-edge protection like CFI. We say this because we hope that a technique (whether it is CFI-based or not) is made such that precise and efficient forward-edge security can be guaranteed; so applying both defenses can complement each other to provide better security. MARDU already provides precise backward-edge CFI via shadow stack, so forward-edge CFI can also be leveraged to further reduce available code-reuse targets.

Note that completely eliminating full-function code reuse and data-oriented programming [94] with low performance overhead and system-wide scalability currently remains an open problem.

3.8 Summary

While current defense techniques are capable of defending against current ROP attacks, most designs inherently tradeoff well-rounded performance and scalability for their security guarantees. Hence, we introduce MARDU, a novel on-demand system-wide re-randomization technique to combat a majority of code-reuse attacks. Our evaluation verifies MARDU's security guarantees against known ROP attacks and adequately quantifies its high entropy. MARDU's performance overhead on SPEC CPU2006 and multi-process NGINX averages 5.5% and 4.4%, respectively, showing that

scalability can be achieved with reasonable performance. By being able to re-randomize on-demand, MARDU eliminates both costly runtime overhead and integral threshold components associated with current continuous re-randomization techniques. MARDU is the first code-reuse defense capable of code-sharing *with* re-randomization to enable practical security that scales system-wide.

Chapter 4

BASTION: Context Sensitive System Call Protection

4.1 Introduction

System calls are a vital part in any program, providing an interface to interact with the operating system (OS). At the same time, they are a core component of various attacks (*e.g.*, arbitrary code execution, privilege escalation) enabling attackers to complete their attack.

Many defense techniques, such as debloating [3, 90, 180, 181, 196], system call filtering [57, 79, 80, 115], and system call sandboxing [105, 133, 223], aim to minimize the available attack surface by disabling unused system calls (*i.e.*, denylist). However, these defenses still allow system calls to be invoked, even if they are used illegitimately, as they remain needed for legitimate use as well.

In this paper, we propose *System Call Integrity*, a novel security policy that enforces the correct use of system calls in a program. The key idea is that a correct use of system calls should follow two properties: *(1) the control flow to the system call when invoked should be legitimate* and *(2) the arguments of the system call should not be compromised*. In order to capture these two properties effectively, we propose three system call contexts, namely Call Type, Control Flow, and Argument Integrity Contexts, which collectively represent how system calls are used in a given program. We enforce the correct use of system calls in a program throughout its runtime in order to break a critical part in attack chains – *i.e.*, illegitimate use of a system call – and negate attacks leveraging system calls. *Call Type Context* allows system calls to be invoked with only the calling convention (*i.e.*, direct call vs. indirect call) they are used within the application. It provides finer-grained system call filtering. *Control Flow Context* only allows system calls to be invoked through a legitimate runtime control-flow path. It can be considered as a scope-reduced CFI, purposely narrow and concentrated to focus on system call related control-flow paths. *Argument Integrity Context* ensures that the arguments of a system call are not compromised. It is data integrity, but only for system call arguments.

In order to enforce our three system call contexts, we propose BASTION, a prototype defense system for system call integrity. To enforce the contexts with minimal runtime overhead, we designed BASTION as a two part system, consisting of a custom compiler pass and a runtime enforcement monitor. Our BASTION compiler pass performs static analysis of system call usages within a program and extracts context information – such as function call types, control-flow paths, and argument tracing – for each system call. Our BASTION runtime monitor uses the compiler-generated metadata and minimal instrumentation to enforce all three system call contexts. Since

system calls are often the most critical point in completing an attack, BASTION intervenes only when a system call is being attempted to be invoked where BASTION confirms if any of the three contexts have been violated.

We evaluate the strength of BASTION's security against both real-world exploits and synthesized attacks of advanced attack strategies [49, 93, 185, 192, 222]. Also, we evaluate our BASTION prototype using varied real-world applications to evaluate its efficiency. Our evaluation confirms that using all three contexts effectively protects sensitive system calls from illegitimate use. BASTION requires minimal instrumentation and narrow runtime interference for their protection, so BASTION imposes negligible performance overhead (0.60%-2.01%) even for system call-intensive applications: NGINX web server, SQLite database engine, and vsFTPd FTP server. Therefore, our BASTION design is lightweight and it is a practical solution to provide comprehensive hardening of system calls.

We make the following contributions in this paper:

- **Novel system call contexts for system call integrity.** We propose three system call contexts, namely Call Type, Control Flow, and Argument Integrity contexts. They collectively represent the legitimate use of a system call in a specific program for system call integrity.
- BASTION **defense enforcing system call integrity.** We design and implement BASTION, which enforces our proposed contexts. The BASTION compiler pass analyzes all system call usage in a program, performs necessary instrumentation, and generates accompanying metadata. BASTION's runtime monitor enforces all static and dynamic aspects of each system call context.
- **Security & performance evaluation.** We conducted security case studies of how BASTION defends against real-world exploits and synthesized attacks with advanced attack strategies. We also performed a performance evaluation with system call intensive real-world applications. Our evaluation shows our contexts can block advanced attacks effectively from illegitimate use of system calls with minimal performance overhead.

4.2 Background

In this section, we first discuss how attackers can weaponize system calls (§ 4.2.1). Then we introduce the current defense techniques and discuss their limitations in securing system call usage in applications (§ 4.2.2).

4.2.1 System Call Usage in Attacks

While an attacker may utilize gadgets or ROP chains within a vulnerable program, their real objective is almost always to reach and abuse critical system calls to achieve arbitrary execution [5, 185, 222]. While there exist over 400+ system calls in recent Linux kernel versions [214], only a certain subset of system calls are actually desired by attackers. These *sensitive system calls* are responsible for critical OS operations, including process control and memory management. Thus, *sensitive system calls* particularly play a key role in attack completion. While recent works [42, 62,

95, 170, 219] have some overlap in sensitive system calls selected, their exact coverage varies.

With insight that a majority of critical attacks need some form of sensitive system call invocation, it raises the question as to why more care has not been taken by recent defense techniques to strengthen defenses around system calls. Current known and future unknown attacks may have different levels of complexity, different approaches, and different goals. However note that almost all must use sensitive system calls to complete their attack. Therefore, it is worthwhile to explore whether strong constraint policies can be created around system calls.

4.2.2 Current System Call Protection Mechanisms

Attack surface reduction. Debloating techniques [3, 90, 180, 181, 196] reduce the attack surface by carving out unused code in a program binary. They leverage static program analysis or dynamic coverage analysis using test cases to discover what code is indeed used. Hence, they can eliminate unnecessary system calls not used in a program. However, many sensitive system calls (*e.g.*, `mmap`, `mprotect`) are used for program and library loading so such sensitive system calls remain even after debloating.

System call filtering. `Seccomp` [111, 115] is a system call filtering framework. A system administrator can define an allowlist/denylist of system calls for an application (*i.e.*, process) and, if necessary, restrict a system call argument to a constant value. However, `seccomp`'s allowlist/-denylist approach cannot eliminate sensitive-but-necessary system calls (*e.g.*, `mmap`, `mprotect`). Moreover, constraining a system call argument to a constant value is applied across the entire application scope so this argument constraining policy could be more permissive than necessary (*e.g.*, an application uses a system call with two very different permissions flags like read-only vs. read-executable).

Control-flow integrity (CFI). Enforcing integrity of control flow can be one way to enforce legitimate use of system calls in a program. CFI defenses [1, 30, 77, 86, 87, 95, 116, 118, 145, 161, 163, 212, 220, 238] aim to enforce the integrity of all forward and backward control flow transfers in a program. CFI-style defenses perform analysis to generate and define an allowed set of targets per-callsite, called an equivalence class (EC). The size and accuracy of ECs are dependent on the analysis technique used to derive legal targets for a given callsite. Imprecise (static) analysis lead to large ECs, allowing attackers to bypass CFI defenses [185, 222]. Enforcing an EC equal to one is ideal – *i.e.*, there is only one legitimate control transfer at a given moment in program execution. However, CFI techniques maintaining an EC equal to one [95, 118] incur high runtime overhead or consume a lot of system resources due to heavy program instrumentation and runtime monitoring (*e.g.*, Intel PT). Even with a perfect CFI technique (*i.e.*, EC=1, low overhead), an attacker can still divert the intended use of a system call by corrupting its arguments (*e.g.*, `pathname` in `execve`, `prot` flag in `mmap`).

Data-flow integrity (DFI). To enforce the integrity of data (*e.g.*, function pointers, system call arguments), DFI [38] tracks data-flow of each and every memory access, instrumenting every `load` and `store` instruction, thus incuring significant performance overhead [73]. Also, the effectiveness of DFI depends on the accuracy of the points-to analysis used to generate the data-flow graph.

4.3 Contexts for System Call Integrity

As discussed previously, debloating and system call filtering make a binary decision whether a system call is allowed to be used. Defining an allowlist or argument values for a program is too coarse-grained and ineffective to protect commonly used sensitive system calls because the context of system call usage is not considered at all. Meanwhile, CFI and DFI are application-wide defense mechanisms, but impose high runtime overhead and require significant system resources. Instead of carrying out fine-grained integrity enforcement, BASTION focuses on protecting an essential component – system call usage – in an attack chain, to thwart attacks.

A legitimate use of a system call should follow two invariants: (1) the control-flow integrity to a system call and (2) the data integrity of system call arguments. In order to enforce these two invariants effectively, we propose three contexts that are established from the system calls themselves. These contexts encompass system calls by incorporating (1) which system call is called and how it is invoked (§ 4.3.1), (2) how a system call is reached within a program (§ 4.3.2), and (3) if its arguments are sound (§ 4.3.3). Collectively, these system call contexts prevent system calls from being weaponized. In stark contrast to prior defense techniques, such as control-flow and data integrity, we selectively protect only program elements relevant to system calls, greatly minimizing invasive flow tracking and runtime overhead. We now describe each context in detail with two real-world code examples (§ 4.3.4).

4.3.1 Call-Type Context

We first propose *call-type context*, which is a per-system call context in a program. With this context, only permitted system calls are able to be called in their allowed manner, either through a direct or indirect call. By applying the call-type context, we are able to provide more fine-grained system call constraints by separating system calls into several sub-categories – (1) *not-callable*, (2) *directly-callable*, and (3) *indirectly-callable* – compared to the binary-decision debloating and system call filtering approaches. The call-type context blocks all unused system calls (*i.e.*, not-callable type). Note, the not-callable type is applied to all system calls, security-critical or not, thus protecting all system calls in a coarse-grained capacity. For allowed system calls, it divides them into ones that can be called from a direct call site (*i.e.*, directly-callable type) and ones that can be called from an indirect call site via a pointer (*i.e.*, indirectly-callable type). In our observation, it is rare for (especially sensitive) system calls to be called from an indirect call site so we can constrain accessibility of indirectly-callable system calls.

4.3.2 Control-Flow Context

Our *control-flow context* enforces that a (sensitive) system call is reached and called only through legitimate control-flow paths during runtime. Hence, it ensures that a control-flow path to a system call cannot be hijacked. All sensitive system calls in a program have their respective valid control-flow paths associated with one another to enforce this context at runtime. This context is specifically narrow to only cover those portions of code that actually reach a sensitive system call;

the remaining unrelated control-flow paths are not considered or covered by this context. Note that our call-type context compliments this context by verifying whether a specific sensitive system call is permitted to be the target of an indirect callsite.

4.3.3 Argument Integrity Context

Our *argument integrity context* provides data integrity for all variables passed as arguments to (sensitive) system call callsites. By leveraging this context, a system call can only use valid arguments when being invoked even if attackers have access to memory corruption vulnerabilities. For this context to be complete, we classify a system call argument type as either (1) a *direct argument* or (2) an *extended argument*. In the direct argument type, the passed argument value itself is the argument (*e.g.*, `prot` flag in `mmap`) so we check the passed argument value for argument integrity. Alternatively, an extended argument type uses one or more levels of indirection of the passed argument for argument integrity checking. For example, in the case of `pathname` in `execve`, we need to check not only the `pathname` pointer but also whether the memory pointed to by `pathname` is corrupted or not, while taking care to avoid time-of-check to time-of-use issues (§4.6.3).

This context takes the expected argument type and its field into consideration for each individual argument for a given system call. For example, in Listing 1, the `path` field of a `ngx_exec_ctx_t` structure (*i.e.*, `ctx->path`) is the one that is verified. In this way, our *argument integrity context* is both a *type-sensitive* and *field-sensitive* integrity mechanism. Moreover, this context includes coverage of all non-argument variables associated with each argument's use-def chain. Thereby, attacks which try to corrupt an argument indirectly are still detected.

Compared to traditional data integrity defenses like DFI [38], we limit the scope to system call arguments (and their data-dependent variables). Further, the argument integrity context checks the integrity of argument values [102] rather than enforcing the integrity of data-flow, thus reducing the runtime overhead.

4.3.4 Real-World Code Examples

We use two code snippets from the NGINX web server [70] as running examples to demonstrate diversity in possible attacks and how all three proposed contexts can negate these attacks collectively. Note that use of `execve` and `mprotect` in Listing 1 and Listing 2, respectively, is legitimate in NGINX, so debloating and system call filtering mechanisms cannot protect these system calls from being weaponized. These attacks only assume the existence of a memory corruption vulnerability such as CVE 2013-2028 (NGINX) or CVE-2014-0226 (Apache).

(1) `execve()`. Listing 1 shows the NGINX function `ngx_execute_proc()`, which legitimately uses the `execve` system call. This NGINX function is intended to update and replace the webserver during runtime. In particular, `execve` is a highly sought after system call for attackers; if it can be reached, attackers may invoke arbitrary code (*e.g.*, shell) and assume control of the victim machine.

There are two attack vectors against the `execve` system call. An attacker can reach the `execve` system call by hijacking the intended control flow (*e.g.*, corrupting a function pointer or return

```
1 // nginx/src/os/unix/ngx_process.c
2 static void ngx_execute_proc(ngx_cycle_t *cycle, void *data){
3     ngx_exec_ctx_t  *ctx = data;
4     // Legitimate NGINX usage of execve system call
5     if (execve(ctx->path, ctx->argv, ctx->envp) == -1) {
6       ...
7     }
8     exit(1);
9 }
10 // nginx/src/core/ngx_output_chain.c
11 ngx_int_t ngx_output_chain(ngx_output_chain_ctx_t *ctx,
12                            ngx_chain_t *in){
13     ...
14     if (in->next == NULL &&
15         ngx_output_chain_as_is(ctx, in->buf) ) {
16         return ctx->output_filter(ctx->filter_ctx, in);
17     }
18     ...
19 }
```

Listing 1: Legitimate use of the `execve` system call in NGINX.

address) if a CFI-style defense is not deployed. Or she can corrupt the system call arguments (*e.g.*, `ctx->path`, `ctx->argv`) to launch a program illegitimately if no data integrity mechanism is used.

This attack scenario leverages an argument-corruptible indirect call site in the victim program such as `ctx->output_filter` (Listing 1, Line 16) in the function `ngx_output_chain()`. The function pointer at this callsite is maliciously redirected to the function `ngx_execute_proc()`, which contains the desired `execve` call. This is followed by corruption of the global `ctx` object, where it is set to attacker controlled values to enable arbitrary code execution.

Our call-type context allows only a direct call of the `execve` system call in NGINX. Also, the control path to `ngx_execute_proc()` is rarely used, only when a runtime update is necessary, so there are limited ways to reach `ngx_execute_proc()`. Our control-flow context enforcement, in tandem with the call-type context, is sufficient to catch such control-flow hijacking of the `execve` system call. Additionally, our argument integrity context can also detect the corrupted system call arguments (*e.g.*, `ctx`) to block this attack proposed by Control Jujutsu [68].

(2) `mprotect()`. Listing 2 is another NGINX code snippet that can be used maliciously. Here, the statement `v[index].get_handler(...)` (Listing 2, Line 6) is found within the function `ngx_http_get_indexed_variable()`. This statement is intended to be used as a generic handler for NGINX indexed variables that selects the appropriate callee from an array of structures with function pointers. Compared to our first attack scenario, `mprotect` does not need to be present within the `v[]` array to be reached.

An attacker can illegitimately reach an illegal offset beyond the `v[]` array by corrupting `index` and thus call the `mprotect` system call without manipulating pointer values. `mprotect` can be weaponized to change the protections of an attacker controlled memory region. To this end, she can corrupt `r` (or `$rsi`) and `v[index].data` (the third argument defining the permission) to **`PROT_EXEC`**`|PROT_READ|PROT_WRITE`.

This attack scenario is able to bypass both code and data pointer integrity (*e.g.*, CPI [126]) as described in the NEWTON attack framework [222]. Instead of corrupting code and data point-

```
1 // nginx/src/http/ngx_http_variables.c
2 ngx_http_variable_value_t *ngx_http_get_indexed_variable( ngx_http_request_t *r, ngx_uint_t index){
3
4     ...
5
6     if (v[index].get_handler(r, &r->variables[index], v[index].data) == NGX_OK) {
7
8         ngx_http_variable_depth++;
9         if (v[index].flags & NGX_HTTP_VAR_NOCACHEABLE) {
10              r->variables[index].no_cacheable = 1;
11        }
12        return &r->variables[index];
13    }
14
15    ...
16
17    return NULL;
18 }
```

Listing 2: Snippet of NGINX code that can be compromised to reach and call the `mprotect` system call elsewhere by corrupting `index` in vulnerable code pointer `v[index].get_handler()`.

ers, the attack relies on finding a callsite that is capable of being manipulated by non-pointer values. Here, the non-pointer variable `index` is manipulated to make the target address of the array `v[index].get_handler` go beyond the array bounds and be redirected to `mprotect`. In addition, this callsite's three arguments `r`, `&r->variables[index]`, and `v[index].data` are also all controllable via non-pointer variables. This allows the callsite to be crafted to an attacker desired function with malicious arguments.

BASTION blocks this attack from the perspective of the system call attempting to be invoked. Our control-flow context in tandem with the call-type context easily detects this control-flow hijacking reaching `mprotect`, whereas CFI and CPI type defenses would not detect this attack. `mprotect` is never invoked indirectly in this application, and is never assigned to `get_handler()`, violating call-type and control-flow contexts. Moreover, our argument integrity context can detect that the leveraged variables `r`, `&r->variables[index]`, and `v[index].data` are illegitimate and never used by any legal system call invocation.

4.4 Threat Model and Assumptions

We assume a powerful adversary with arbitrary memory read and write capability by exploiting one or more memory vulnerabilities (*e.g.*, heap or stack overflow) in a program. We assume that common security defenses – especially Data Execution Prevention (DEP) [110, 148] and (coarse-grained) Address Space Layout Randomization (ASLR) [210] – are deployed on the host system. Hence attackers cannot inject or modify code due to DEP. We also assume a shadow stack will be employed (*e.g.*, CET [195]) as this technology is mature [32, 53] and is now available in hardware [130]. We assume that the hardware and the OS kernel are trusted, especially `secomp-BPF` [115] and the OS's process isolation. Attacks targeting the OS kernel and hardware (*e.g.*, Spectre [121], Row Hammer [120]) are out of scope.

We focus on thwarting a class of attacks exploiting one or more (sensitive) system calls in their

System Call Classification	Applicable System Calls
Arbitrary Code Execution	`execve, execveat, fork, vfork, clone, ptrace`
Memory Permissions	`mprotect, mmap, mremap, remap_file_pages`
Privilege Escalation	`chmod, setuid, setgid, setreuid`
Networking	`socket, bind, connect, listen, accept, accept4`

Table 4.1: Classification of sensitive system calls commonly leveraged by attackers. We classify each *security-critical* system call with the attack vector that commonly abuses it.

attack chain. This is because core attacker goals are to issue one or more (sensitive) system calls. System calls are the primary means to interact with the host OS. Attacks are therefore ineffective if they do not have access to these system calls [35]. For example, Göktas et al. [84] need to change the permissions of an existing memory area (*e.g.*, `mprotect`) to achieve their attack goals. Therefore BASTION protects a subset of available system calls (Table 4.1) that can allow an attacker escape beyond the scope of an application to obtain control over the host system. BASTION concentrates on a subset of system calls as not all system calls perform meaningful security-critical actions. Likewise, BASTION protects a subset of data connected to system calls, instead of all program data.

4.5 BASTION Design Overview

BASTION aims to strengthen the integrity surrounding sensitive system calls by enforcing the three proposed contexts (*i.e.*, Call-Type, Control-Flow, and Argument Integrity), such that attacks leveraging system calls can be mitigated. Similar to prior defenses [42, 62, 77, 95, 170, 220], we choose 20 sensitive system calls (Table 4.1). We chose these system calls such that BASTION can successfully defend against attacks that achieve arbitrary code execution, memory permission changes, privilege escalation, or rogue network reconfiguration. Note, this list can be easily extended to include other system calls.

As illustrated in Figure 4.1, BASTION consists of two core components: (1) a BASTION-compiler pass, which performs context analysis, instrumentation, produces a BASTION-enabled binary, and generates context metadata, and (2) a BASTION monitor, which enforces system call contexts during runtime. We now explain BASTION's compiler (Section 4.6) and runtime monitor (Section 4.7) in detail.

4.6 BASTION Compiler

Our BASTION compiler derives the necessary information for proper enforcement of our system call contexts. BASTION also leverages a light-weight library API for dynamic tracking of sensitive data related to the argument integrity context to properly track relevant system call arguments. We now describe our program analysis for each context and creation of a BASTION-enabled binary.

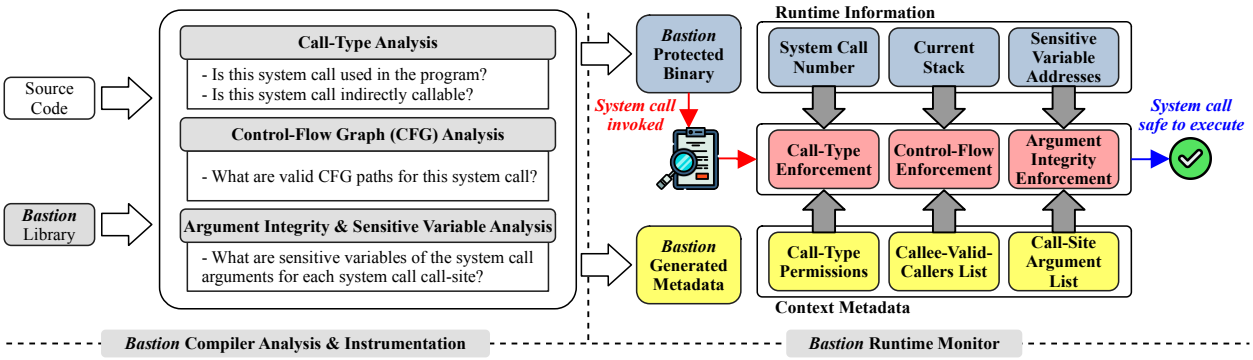


Figure 4.1: Design overview of BASTION. At compile time, BASTION analyzes and generates a program's context metadata. For call-type and control-flow contexts, BASTION generates static metadata. For the argument integrity context, BASTION generates metadata for static argument values (*e.g.*, constants) and instruments the program to enable tracking of dynamic argument values. At runtime, the BASTION monitor hooks on all invocations of sensitive system calls and verifies all three contexts of the system call.

4.6.1 Analysis for Call-Type Context

To enforce the call-type context, the BASTION compiler analyzes a program and classifies system calls in the program into three categories: (1) not-callable, (2) directly-callable, and (3) indirectly-callable types, as discussed in § 4.3.1. It analyzes the entire program's LLVM IR instructions and checks all call instructions. If a system call is a target of a direct function call, the BASTION compiler classifies it into a directly-callable type. If the address of a system call is taken and used in the left-hand-side of an assignment, that system call can be used as an indirect call target, and the BASTION compiler classifies it as an indirectly-callable type. Note that a system call can be both directly-callable and indirectly-callable. All other arbitrary system calls not belonging to either type are set as not-callable. Any attempt by an attacker to reach a not-callable system call, security-sensitive or not, is not permitted by BASTION.

After this analysis is completed, the BASTION compiler generates metadata which contains, (1) pairs of system call numbers and their call type, and (2) a list of *legitimate* indirect callsites (*i.e.*, offset in a program binary). This metadata is used by the BASTION runtime monitor to enforce the call-type context.

4.6.2 Analysis for Control-Flow Context

BASTION uses the control-flow context to prevent control-flow hijacking attacks that illegitimately reach system calls. Enforcement of the control-flow context is performed *only when a system call is invoked*. Our approach is different from conventional CFI techniques, which enforce the integrity of *every indirect control-flow transfer*.

BASTION analyzes a control-flow graph (CFG) of the entire program to identify all function *callee→caller* relationships that reach system call callsites. The `llvm-link` utility is used to merge

API	Description
`ctx_write_mem(p,size)`	Update the shadow copy of `p` in `size`
`ctx_bind_mem_X(p)`	Bind a memory `p` to `X`-th argument
`ctx_bind_const_X(c)`	Bind a constant `c` to `X`-th argument

Table 4.2: BASTION library API for argument integrity context.

all source code into a single IR module and create the CFG. For each system call callsite in the CFG, the BASTION compiler recursively records all callee→caller associations. Recursive analysis stops once reaching either `main()` or an indirect function call. The BASTION runtime monitor verifies callee→caller relations until the bottom of the stack (*i.e.*, `main`), or an indirect callsite, by unwinding stack frames.

With the call-type context and control-flow context in tandem, BASTION verifies that control-flow reaching a system call follows (1) legitimate direct callee-caller relations and (2) is a legitimate indirect call from a valid indirect callsite. Once analysis for control-flow context is done, BASTION generates metadata which consists of the pairs of callee and caller addresses in a program binary.

4.6.3 Analysis for Argument Integrity Context

Lastly, the BASTION compiler analyzes a program to enforce argument integrity for system calls. Note that this is the only context that requires instrumentation. We now discuss what variables should be protected (Section 4.6.3), how to check if an argument is compromised (Section 4.6.3), how our instrumentation achieves dynamic tracking of arguments (Section 4.6.3), and what metadata is generated (Section 4.6.3).

Protection Scope

In order to properly enforce argument integrity, BASTION needs to check not only system call arguments but also an arguments' data-dependent variables. We collectively call these protected variables as *sensitive variables*. Figure 4.2 shows how BASTION enforces the argument integrity context for each of the arguments in the `mmap` system call. At Line 22, `NULL`, `-1`, and `0` are constant arguments; `gshm->size`, `prots`, and `b2` are memory-backed arguments; `flags` is a data-dependent sensitive variable, which is passed from the function `foo`.

Checking Argument Integrity

In order to verify the argument integrity for each system call, BASTION adopts data value integrity [103] for each sensitive variable. BASTION maintains a shadow copy of the sensitive variable's legitimate value in a shadow memory region and updates the shadow copy whenever the sensitive variable is updated legitimately. Before a system call is invoked, BASTION binds each

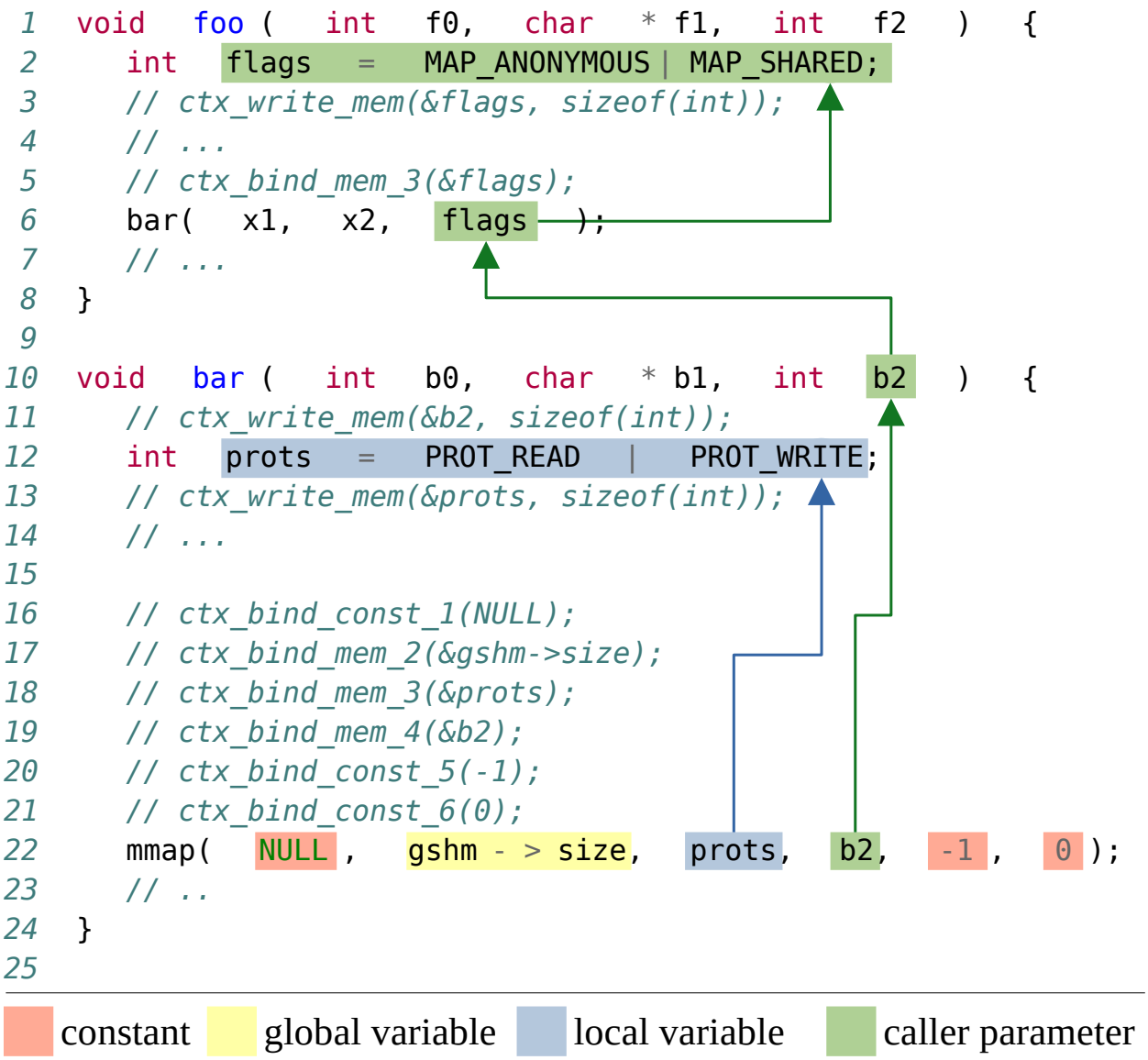


Figure 4.2: BASTION instrumentation for argument integrity. Protection of memory-backed arguments is augmented using field-sensitive use-def analysis (`size` field of `gshm`, `b2`←`flags`) at an inter-procedural level.

argument to a certain position for the system call so the BASTION runtime monitor can check argument integrity using the BASTION-maintained shadow copy of the respective sensitive variable.

The BASTION runtime library provides the API shown in Table 4.2. `ctx_write_mem(p,size)` updates the shadow copy of a sensitive variable located at address `p` with `size`. Note that a constant argument (*e.g.*, `NULL`) does not need to have a shadow copy, because its legitimate value is determined statically at compile time. Hence `ctx_write_mem` is only used for memory-backed sensitive variables (*e.g.*, `gshm->size`, `prots`, `b2`, and `flags` in Figure 4.2). For argument binding, `ctx_bind_mem_X(p)` binds a memory-backed sensitive variable at `p` to the `X`-th argument of the

associated system call callsite. Similarly, `ctx_bind_const_X(c)` binds the constant value `c` to the `X`-th argument of the callsite. Note that binding is applied to both system call callsites as well as callsites passing sensitive variables (*e.g.*, `bar()` callsite, Line 6 in Figure 4.2). We note that it is not necessary to instrument if an argument is a direct or extended argument as such a distinction is system call & position-specific. Instead, BASTION's monitor can recover the system call being verified so we design the monitor to handle this distinction, further minimizing instrumentation. For example, if BASTION's runtime monitor discerns `execve` is being verified, it knows the first argument of `execve` is an extended argument, and thereby will automatically verify not only the pointer value but also pointee memory contents for only the first argument, while the remaining arguments will be verified as direct arguments. To add, because the list of sensitive system calls is short, it is easy to specialize the rules for these arguments.

Sensitive Variable Instrumentation

At a high level, the BASTION compiler performs field-sensitive, inter-procedural analysis to identify all sensitive variables, including *data-dependent variables*. To identify all sensitive variables, the BASTION compiler analysis performs three steps. First, it enumerates all variables used in system call arguments. These variables are the initial set of sensitive variables. Second, it performs a backward data-flow analysis, traversing the use-def chains to derive any other variables used to define sensitive variables. Such newly identified data-dependent variables are added to the set of sensitive variables. Third, if there is a write to a field of a struct (*e.g.*, `size` field of `gshm` in Figure 4.2), that write is added to the sensitive variables. Analysis repeats the second and third steps until no new sensitive variables are found. Note this field-sensitive analysis is also effective for tracking sensitive global and heap variables.

BASTION first follows and instruments the callsite's system call argument variables via intra-procedural analysis. If an argument variable is updated by a caller function's parameter (*e.g.*, `b2←flags` in Figure 4.2), BASTION adds the caller parameter to the sensitive variables and performs inter-procedural analysis between the caller and callee for the caller's parameter (*e.g.*, `flags` in Figure 4.2). This process is done recursively until an origin for the sensitive variable's use-def chain is found.

Once all sensitive variables are identified, BASTION instruments `ctx_write_mem` after any memory-backed sensitive variable `store` to keep its shadow copy up-to-date. Before each sensitive system call callsite, BASTION instruments `ctx_bind_mem_X` or `ctx_bind_const_X` to bind an arguments to their respective argument position `X`. Regarding the analysis of extended arguments, BASTION instruments all possible use-def chains. While this may incur more instrumentation, it does not lower security guarantees and we did not observe any additional performance overhead. In practice, the call depth to set system call arguments is fairly shallow – within the same function or only a few functions away.

We note that our instrumentation does not impose any control transfers to the monitor and all updates remain in userspace. Even if an attacker happens to skip our instrumentation, since non-system call relevant control-flow is not monitored or enforced by BASTION, malicious intent can still be detected by BASTION. In this case, legitimate system call argument values will not have

been properly updated to their expected values.

BASTION-compiler Generated Metadata

In the case of argument integrity context metadata, the compiler specifies an entry for each sensitive system call callsite. Each callsite entry includes the callsite file offset and the argument types (*i.e.*, constant vs. memory-backed arguments, direct vs. extended arguments). For a constant argument, the expected value is recorded as well. For a memory-backed argument, the argument index denotes that a bound value should be retrieved from the BASTION shadow memory region and compared with the value in an argument register (*e.g.*, `$rdi`, `$rsi`, `$rdx`).

4.7 BASTION Runtime Monitor

The BASTION monitor enforces all three of our proposed system call contexts at runtime. We specifically design the BASTION monitor as a separate process from the application being protected. In doing so, attackers are unable to avoid BASTION's runtime hooks that occur at *sensitive* system call invocations. We now describe how the monitor is initialized and how each context is enforced.

4.7.1 Initializing the BASTION Monitor

Loading metadata. The BASTION monitor is invoked with a target application binary path. The monitor retrieves `ELF` [136], `DWARF` [147], and linked library information to recover symbol addresses. It then loads BASTION context metadata into the monitor's memory.

Launching a BASTION-protected application. After loading metadata, BASTION performs `fork` to spawn a child process where the child runs the BASTION-enabled application after synchronization is setup by the monitor. Specifically, BASTION initializes a shadow memory region under a segmentation register (`$gs` in Linux) for shared use between the application process and the BASTION monitor process. BASTION also initializes `seccomp` [111, 115] to trap on sensitive system calls in the child process and `ptrace` [137] to access the application's state (including the shared shadow region attached to the application's virtual address space). Once both the target application and BASTION are synchronized, the BASTION monitor allows the application to proceed and begins monitoring. Then, any sensitive system call invocation attempt will trap the BASTION monitor to perform context integrity verification by poking the application's execution state, before allowing the system call to be executed. Otherwise, BASTION monitor rests in an idle state until the next sensitive system call is invoked.

Trapping a system call invocation. The BASTION monitor initializes a custom `seccomp-BPF` filter to trap on the application's sensitive system call invocations using the call-type metadata. `SECCOMP_RET_ALLOW` is specified for all non-sensitive system calls such that invoking a them does not trap the monitor. `SECCOMP_RET_KILL` is specified to disable any not-callable system calls. Finally, `SECCOMP_RET_TRACE` is specified for directly- and indirectly-callable system calls such that

these system calls can be verified by the BASTION monitor. Note that a child process or thread spawned by a BASTION-protected application has the same `seccomp` policy as its parent process is protected by the same BASTION monitor process.

Accessing application state and shadow memory. The BASTION monitor needs to retrieve an application's runtime information and its shadow memory to verify integrity of the system call contexts before allowing the execution of a sensitive system call. The BASTION monitor retrieves current register values and system call number via `ptrace`. It uses `process_vm_readv` [112] to access arbitrary memory in the application's address space, including stack, heap, and the shadow memory region.

As briefly discussed, shadow memory is initialized when a BASTION-protected application is launched and belongs to the application's address space. It is an open-addressing hash table maintaining a shadow copy (*i.e.*, legitimate value) of a sensitive variable and argument binding information for the argument integrity context. The key to access this hash table data is an address; the sensitive variable address and callsite address are used to access its shadow copy and argument binding information, respectively. Note BASTION's shadow memory region relies on sparse address space support of the underlying OS like metadata store designs in prior studies [103, 126].

4.7.2 Enforcing Call-Type Context

For call-type context, the BASTION monitor retrieves the system call number and the program counter (`rip`) where the system call is invoked using `ptrace`. The BASTION monitor uses the `rip` to retrieve and check the call type of the callsite from its metadata. For example, the BASTION monitor decodes the `call` instruction at Line 22 in Figure 4.2 using `rip` to determine if the `mmap` call was made direct or indirect. If the call type is allowed, BASTION continues to the next context check. Otherwise, the BASTION monitor assumes this is an attack attempt and immediately kills the protected application and gracefully exits.

4.7.3 Enforcing Control-Flow Context

For control-flow context, the BASTION monitor retrieves a copy of the current stack trace when a system call attempt is made and verifies that the current call stack adheres to the CFG metadata, represented as a list of callees and their respective valid callers. The BASTION monitor unwinds the retrieved stack frame to get each function callsite offset. It then checks if the unwound caller is in the valid caller list of the callee in the CFG metadata. For example, in Figure 4.2, function `foo()` is a valid caller of function `bar()`. This is iteratively performed until the entire stack has been vetted or an indirect call is encountered. When handling an indirect call, BASTION ends verification at this point and verifies the partial stack trace encountered matches the expected one derived at compile time. Thus, there are no false-positives or false-negatives. If a mismatch in a control-flow transition is found, the BASTION monitor assumes this is a control-flow hijacking attempt to illegitimately reach this system call, and kills the application, as done for the call-type context.

4.7.4 Enforcing Argument Integrity Context

For argument integrity context, the BASTION monitor verifies integrity of all sensitive variables in the current call stack. Take Figure 4.2 as an example. At function `bar()`'s stack frame, the BASTION monitor verifies all bound constant variables (*i.e.*, `NULL`, `-1`, `0`) and memory-backed variables (*i.e.*, `gshm->size`, `prots`, `b2`). Once checking the current stack frame is done, it unwinds the stack and verifies function `foo()`'s sensitive variable (*i.e.*, `flags`). For each callsite, the BASTION monitor uses the current `rip` value to retrieve the associated argument integrity context metadata reporting types of each argument (constant vs. memory-backed arguments) to know how to verify it. Additionally, recall from Section 4.6.3, the monitor automatically processes each argument as direct or extended as needed. Particularly for non-system call callsites, BASTION also uses argument integrity context metadata to determine which arguments need to be verified. For each callsite's sensitive argument, the BASTION monitor compares the register (actual) argument value to the BASTION-traced value retrieved from shadow memory. BASTION iteratively traverses all function frames currently on the stack. If values match, the arguments are legitimate and the system call can proceed. Otherwise, the BASTION monitor kills the application and concludes runtime monitoring.

4.8 Implementation

We implemented BASTION on Linux x86-64 v5.19.14. BASTION analysis and instrumentation is implemented as an LLVM Module pass in 3,939 lines of code (LoC). BASTION's C runtime library (659 LoC) implements memory management functions to enable the monitor to read and bind arguments throughout the entire stack frame. All library functions are inlined to maximize performance. The BASTION runtime monitor is a C-program (7,313 LoC). It leverages `seccomp-BPF` and `ptrace` for triggering system call enforcement. BASTION also employs CET [195] a hardware-based shadow stack by specifying compiler flag `-fcf-protection=full`. Intel Tiger Lake and AMD Ryzen 7 processors onwards [130] with `Glibc` v2.28+ [129, 143], `Binutils` v2.29+ [204], and Linux kernel v5.18+ fully support CET.

4.9 Evaluation

In this section, we evaluate how effective BASTION's system call contexts are. We measure the performance overhead of BASTION with three real-world applications: the NGINX web server [70], the SQLite database engine [51], and vsftpd [44], an FTP server. Our evaluation includes partitioning the individual costs of each BASTION system call context, examining the cost-benefit analysis for each as well as reporting static and runtime statistics.

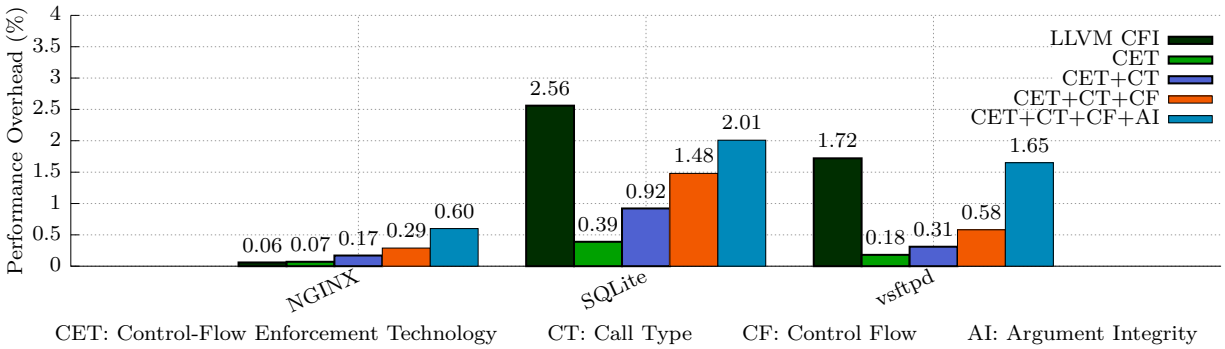


Figure 4.3: Performance overhead of BASTION for NGINX, SQLite, and vsftpd. All values are compared to an unprotected baseline vanilla version. For reference to state-of-the-art, we also include the individual overhead for (coarse-grained) LLVM Control-Flow Integrity (LLVM CFI) [209] and Control-Flow Enforcement Technology (CET) [195].

4.9.1 Evaluation Methodology

Evaluation setup. We ran all experiments on an 8-core (16-hardware thread) machine featuring an AMD Ryzen 7 PRO 5850U processor and 16 GB DDR4 memory. All benchmarks were compiled with the BASTION LLVM compiler. Results are reported average over five runs.

Applications. We ran NGINX, SQLite, and vsftpd as these real-world applications are widely deployed and as such, they are often victim to attack. Also, these are I/O-intensive and system call-heavy applications.

4.9.2 Performance Evaluation

For the performance evaluation of BASTION, we report both the relative performance overhead compared to the unprotected baseline version (Figure 4.3) as well as the raw performance numbers of all three applications (Table 4.3). We ran all evaluations with Address Space Layout Randomization (ASLR) [210] enabled as BASTION is based around relative-addressing and has no incompatibility issues with ASLR-based defenses. Before applying BASTION, we enabled CET and compared the runtime overhead compared to the baseline version. For all three applications, we observe CET incurs negligible overhead. Finally, we observed that the initialization cost of BASTION monitor is always negligible (on the order of ten to twenty milliseconds, *e.g.*, ≈21 ms for NGINX).

NGINX. To test NGINX, we use `wrk` [83], a HTTP benchmarking tool. `Wrk` sends concurrent HTTP requests to a web server and measures throughput. We place the `wrk` client on a different machine, but on the same local network as the NGINX webserver. NGINX is configured to handle a maximum of 1,024 connections per processor and have 32 worker threads. We measure throughput for a 20-second run. `wrk` spawns the same number of threads as NGINX's configured worker count where each `wrk` thread generates HTTP requests for a 6,745-byte static webpage.

The runtime overhead for NGINX minimally increased as each BASTION context was enabled. The

Application	Unprotected Vanilla	Runtimes with Hardware/Software Mitigations				
		LLVM Control-flow Integrity (LLVM CFI)	Control-flow Enforcement Technology (CET)	CET+DI	CET+DI+CF	CET+DI+CF+AI
NGINX (MB/sec)	110.61	110.54	110.52	110.42	110.28	109.94
SQLite (NOTPM)	37,107.41	36,156.15	36,961.91	36,764.50	36,560.02	36,360.85
vsftpd (sec)	10.75	10.93	10.77	10.79	10.81	10.93

Table 4.3: Benchmark numbers for NGINX, SQLite, and vsftpd on which Figure 4.3 is based on. We measure NGINX's throughput of data (MB/s). SQLite is evaluated using DBT2 [164] which records New Order Transactions Per Minute (NOTPM). Lastly, for vsftpd, we measure seconds elapsed to download a 100 MB file. For NGINX and SQLite, higher is better, whereas lower is better for vsftpd.

overhead when applying full BASTION protection (all three contexts enabled, Call-Type, Control-Flow, & Argument Integrity) never incurred more than 0.60% degradation compared to the unprotected NGINX baseline. Figure 4.3 shows this breakdown in more detail. In NGINX, protecting the subset of 20 sensitive system calls with just the monitor adds barely ≈1% overhead. As expected, the Argument Integrity context adds the most overhead following the monitor itself since this context is the most complex.

Analysis of NGINX source code reveals that NGINX utilizes a vast majority of sensitive system calls (*e.g.*, `mprotect`, `mmap`) during its initialization phase while seldom using any sensitive system calls when idle or processing requests. This results in BASTION rarely being triggered during runtime. Table 4.4 shows combined initialization and runtime statistics of all (sensitive) system calls invoked during benchmarking; it reveals that only a call to `accept4` is made per-request and dominates the system call invocation count. Additionally, evaluating NGINX showed that for all system call invocations (not including those originating from a library), the average call-depth is only 5.2 frames, with 4 and 9 being minimum and maximum stack call-depths encountered, respectively.

SQLite. SQLite [51] is a widely deployed, transactional SQL database engine. To test the performance throughput of SQLite, we use the DBT2 [164] database transaction processing benchmark. DBT2 mimics a mix of `read` and `write` SQL operations for large data warehouse transactions. We ran DBT2 with its default configuration with a 10 second new thread delay and a 10 minute workload duration. We measure DBT2's number of new-order transactions per-minute (NOTPM) for performance.

Figure 4.3 shows the BASTION performance breakdown for each context. Adding Call-Type and Control-Flow context checking increases BASTION's overhead to 0.92% and 1.48%, respectively. Subsequently, adding Argument Integrity context checking for SQLite's system calls brings BASTION's overhead to 2.01%. As in NGINX, Argument Integrity context enforcement contributes the most overhead, relative to Call-Type and Control-Flow contexts. However, note that since our Argument Integrity context strategically only requires tracing and enforcing value integrity for select sensitive variables used as system call arguments, this contexts performance overhead is magnitudes smaller than overhead imposed by conventional application-wide DFI-style defenses.

SQLite largely uses system calls to initialize and setup its worker threads based off the DBT2 configuration. We expect overhead is further amortized for SQLite in a long standing real-world application setting. Table 4.4 shows system calls invoked during SQLites runtime. The contrast

in Argument Integrity costs can be attributed to differences in system call usage and argument patterns. SQLite relies more on `mprotect` compared to NGINX or vsftpd.

VSFTPD. vsftpd is evaluated using `dkftpbench` [52], an FTP benchmark program. `dkftpbench` simulates users downloading files from a FTP server. We place the `dkftpbench` client on the same machine as BASTION protected vsftpd. `dkftpbench` is configured to fetch a `100 MB` file from vsftpd launching clients one after another for a 120 second duration. All other options are left as default.

`Dkftpbench` showed negligible runtime overhead for all BASTION contexts compared to the unprotected baseline. In the worst case, BASTION incured 1.65% overhead.

Table 4.4 substantiates this performance overhead, showing that vsftpd invokes system calls far fewer times than either NGINX or SQLite. Similar to NGINX, `accept` accounts for most of BASTION monitor checks. When designing BASTION to support `accept` and `accept4`, we noted that only these two system calls had a more complex argument (*e.g.*, `struct sockaddr`) compared to other system call arguments. We therefore optimized the BASTION monitor to support verifying this structure in a specific way to improve argument integrity runtime.

System call statistics. Table 4.5 shows instrumentation statistics regarding BASTION including the total number of sensitive system call callsites and each BASTION API instrumentation count. This table shows that sensitive system calls have a very small footprint even in large production applications. Specifically, note the drastic difference between the number of application callsites (Table 4.5, Row 1) compared to the number of sensitive system call callsites (Table 4.5, Row 4). Consequently, BASTION's instrumentation footprint is also relatively small, with a maximum of 5,287 instrumentation points in NGINX. A key finding is that in all three real-world applications, sensitive system calls are *never* legitimately called indirectly via a function pointer (Table 4.5, Row 5). This facet allows BASTION to entirely invalidate attack schemes that rely on reaching system calls from a corrupted pointer.

Comparison against CET and LLVM CFI. We also compared BASTION with other state-of-the-art defenses, specifically LLVM CFI (forward edge protection) and CET (backward edge protection). Results are depicted in Figure 4.3 and Table 4.3. Note that we were not able to simultaneously enable CET with LLVM CFI as LLVM CFI does not function properly when paired with CET.

CET works by maintaining a secondary (shadow) stack that cannot be directly modified by applications. Upon returning from a function, the CPU compares return addresses in the shadow stack and the normal stack. If these two addresses differ, a protection fault is raised. Our results show that CET incurs negligible overhead ($< 0.5\%$).

LLVM CFI [209] is a coarse-grained CFI technique. Our results show that LLVM CFI incurs low performance overhead ($< 3\%$). LLVM CFI performs verification at every indirect callsite while BASTION is designed to only perform verification at sensitive system call invocations. LLVM CFI checks indirect and virtual function calls only using function type information. Since it is not fine-grained, it does not guarantee a unique target as in recent CFI works [95, 118]. Moreover, CFI does not have such a concept to check whether a function is allowed to be called directly or indirectly as done for our Call-Type context, nor does CFI verify argument integrity compared to BASTION.

Therefore Call-Type and Control-Flow contexts cannot be directly replaced with CFI.

Summary. Our evaluation confirms our insights of sensitive system call usage patterns being used sparsely. For all real-world applications tested, we see BASTION instrumentation and monitoring for all three contexts incurs low (<3%) performance degradation. Compared to related defense strategies, we are on par or better (*e.g.*, 7.88% - uCFI [95], 7.6% - OS-CFI [118], 2.7% - OAT [206]).

4.10 Security Evaluation

We conducted case studies with 32 attacks in Table 4.6, which include ROP payloads, real-world CVE exploits, and synthesized attacks recent attack strategies [49, 93, 185, 192, 222]. The results confirm that even if one context is bypassed, another context in BASTION can compensate and still prevent the attack vector from being viable.

4.10.1 ROP Attacks

While CET is now available on the newest processors from Intel and AMD, older processors do not have a built-in shadow stack defense mechanism. We now explain how BASTION can defend ROP in the absence of CET.

ROP payloads work by stitching together various ROP-gadgets present in a victim application to create their attack. Generally, they will leverage one or more system calls (*e.g.*, `execve`, `mprotect`) to complete the attack.

ROP payloads can execute user (non-`root`) commands by leveraging the `libc` library call `system` (which internally calls `fork` and `execl` system calls), and passing the arguments `"/bin/sh"`. Or they can directly leverage an `exec`-type system call, to create access to a `root` shell. ROP payloads can also manipulate memory permissions and abuse a memory corruption vulnerability by using `mprotect` or `chmod` system calls to change memory or file permissions to be executable (*e.g.*, `RWX`) in order to setup their attacker directed script, before pivoting the stack or code pointer to the start location.

If the victim application does not use these system calls, BASTION blocks these invocations with the Call Type context. Or if these ROP payloads advance by directly manipulating control-flow and variables to reach and prepare these system calls, including setting flags to uncommonly used natively by applications (*e.g.*, making a memory region executable), Control Flow or Argument Integrity contexts will detect these attacks.

4.10.2 Direct Attacker Manipulation of System Calls

In this attack strategy, attacks go after system calls directly, crafting attacks that setup callsites and arguments to desired values via code pointer and variable corruption.

The CsCFI attack leverages `mprotect` to make the entire `libc` readable, writable, and executable, revealing the code layout to perform arbitrary code execution. AOCR's Attack 1 instead leverages `open` and `write` to reveal the code layout of NGINX to execute arbitrary code. In both cases, BASTION's Call-Type context blocks these attacks. In the CsCFI attack, `mprotect` is never used by the application. In Attack 1, `open` is legitimately used in NGINX, but only ever as a direct call. Moreover, BASTION records all valid call-traces for sensitive system calls as well as instrumenting all arguments for sensitive system calls, allowing BASTION's Control-Flow and Argument integrity contexts to also detect the anomaly of being called from an unexpected callsite with untraced arguments.

Several real-world CVE's also fall under this category. While they each individually affect different applications, they all rely on inherent memory corruption vulnerabilities and abuse them to maliciously manipulate program data including code pointers and argument variables. Therefore, BASTION can address all these CVE's because they aim to achieve arbitrary code execution by corrupting code pointers and arguments to point to system calls that are either never used or only used via direct callsites.

In comparison to BASTION, LLVM CFI cannot defend against either attack. Since LLVM CFI employs a coarse-grained defense scheme, it only enforces that function calls match the static type used at the callsite [209]. In the CsCFI attack, even though `mprotect` is never used by the application, its address is still taken as this system call is necessary to support dynamic loading of shared libraries. Related, AOCR Attack 1 leverages code pointers whose type matched to system calls `open` and `write`, allowing these control-flow violations to bypass LLVM CFI.

4.10.3 Indirect Attack Manipulation of System Calls

In this attack strategy, attacks evade popularly deployed defense strategies such as CFI, CPI, and system call filtering. Such strategies include attacks like full-function code re-use [68], data-oriented attacks [40, 94], and COOP [192].

The NEWTON CPI attack avoids corrupting any code or data pointers. It corrupts the index variable of an array of function pointers to make the array index point to a system call location. BASTION is able to defend against this attack with all three contexts. The Call-Type context blocks the invocation of a system call never used in the program code base. Similarly, Control-Flow and Argument Integrity context will not have legitimate system call information for the corrupted callsite or arguments used thereby thwarting the attack as well.

The AOCR Apache attack finds an indirect code pointer of an `exec` function in `ap_get_exec_line()`. Because there is a legitimate indirect call to `exec`, BASTION's Call-Type context will be bypassed. Instead, the hijacking of `ap_get_exec_line()` onto a corruptable function pointer is what gives away the attack, and allows BASTION to block it with the Control-Flow context.

Lastly, AOCR NGINX Attack 2 [185], COOP [192], and Control Jujutsu [68] leverage inherent memory vulnerabilities as well as the inherent program control-flow against itself to gain hijack control.

Compared to BASTION, LLVM CFI cannot defend against this attack strategy – AOCR NGINX

Attack 2, COOP, or Control Jujutsu – as these attacks specifically leverage legitimate control-flow transfers to reach security sensitive system calls. For instance, COOP starts with a buffer over-flow filled with fake counterfeit objects by the attacker and leverages virtual C++ functions that would appear as benign execution to LLVM CFI by not violating type. In AOCR NGINX Attack 2, with the help of the attacker-controlled worker, the attacker only needs to corrupt global variables to cause NGINX's master process loop to call `exec` under attacker chosen parameters.

In this case, Call-Type and Control-Flow contexts will seem legitimate for BASTION. However, these attacks still need to corrupt system call arguments to attacker directed values, thereby allowing BASTION to detect and block these attack vectors. Compared to LLVM CFI, BASTION employs the Argument Integrity context and is therefore tighter in its constraints compared to CFI. For this reason, BASTION is able to defend against indirect attack manipulation strategies.

4.11 Discussion and Limitations

4.11.1 BASTION under Arbitrary Memory Corruption

In theory, a powerful adversary with arbitrary memory `read/write` capability can circumvent all three of BASTION's contexts. However, in practice it will be very challenging particularly because our analysis pass is able to *statically* determine most constraints – direct vs. indirect call, legitimate stack traces, and constant arguments – to successfully enforce correct system call invocation in a program. For example, if `mprotect()` is used only with a constant value, `PROT_READ`, in a program, then it is impossible to call `mprotect()` with `PROT_EXEC` because such static constraints are maintained by the monitor processor running in a separate address space; this data is never available to the protected application. To bypass all three of BASTION's contexts, the attacker realistically would need to perform arbitrary `read/write` many times to match the expected context values without violating static constraints (*e.g.*, constant arguments). This is very challenging to carry out in real attack scenarios, thus BASTION raises the difficulty to complete such an attack.

4.11.2 Protecting Filesystem Related System Calls

One promising extension of BASTION is protecting file system related system calls to prevent information disclosure attacks. The main challenge is that this type of system call is called much more frequently than BASTION's sensitive system calls. To see the feasibility of such an extension, we extended BASTION's coverage to include all file system related system calls and their variants.

We present the performance overhead of this extension compared against the unprotected baseline in Table 4.7. Full BASTION context checking (Row 3) incurs high overhead – *e.g.*, 96.7% for NGINX. To understand where this overhead comes from, we break down BASTION into three sub-steps: 1) just hooking system calls (Row 1) to measure seccomp overhead, 2) fetching the protected program's information using `ptrace` (Row 2), and 3) verification of BASTION's three contexts (Row 3). Our results show overhead of hooking system calls ($< 0.29\%$, Row 1) and full context checking ($< 0.82\%$, delta between Rows 2 and 3) is negligible. A majority of overhead results from

fetching protected process state using `ptrace` (< 95.7%, delta between Rows 1 and 2) due to the additional context switching overhead to access the protected program.

One approach to (almost) completely eliminate `ptrace` overhead would be to run the BASTION monitor inside the kernel as either a kernel module or via eBPF code. This solution would completely resolve overhead incurred from context switching; by being within the kernel, the BASTION monitor would now have transparent access to the protected application's process state to retrieve register values, etc. With the optimization of replacing `ptrace` with in-kernel execution, we can extend BASTION's system call protection scope with low performance overhead. Ultimately, BASTION can used as a foundational platform and be extended to implement defense for a range of threat models in the future, such as information disclosure.

4.11.3 Impact of Not Protecting of All System Calls

We designed BASTION to deeply protect a subset of system calls that have security implications (*i.e.*, sensitive system calls). That being said, BASTION's Call-Type context can still sort out all not-callable system calls in a program (§ 4.3.1) so they cannot ever be weaponized. If an arbitrary (sensitive or not) system call is never used by an application, the BASTION monitor can still enforce the Call-Type context and disallow any use.

Not all system calls are security-critical (*e.g.*, `getpid`) as they do not perform any operation that can adversely affect the host system for attacker gains. Even if uncovered, non-sensitive system calls are used in an attack chain, an attacker should still exploit at least one sensitive system call. In this way, BASTION is able to block the attack by detecting unintended usage of a sensitive system call.

However, we do not claim that our sensitive system call selection is "perfect". Presently, there is no consensus of how to quantitatively assign a system call's "danger level". Thus far, this ongoing effort has been empirically deduced from performing case studies and examining real-world CVEs. Explicit system call classification is sparse in literature; Bernaschi et al. [21] provides some guidance, however their analysis is limited to buffer overflow-based attacks. More recent work ranks system call risk based on using information retrieval techniques in an exploit code analysis [106]. More comprehensive analysis remains an interesting future direction.

4.12 Related Work

Since we already discussed the most closely related work in Section 4.2, we discuss other related studies in this section.

Advanced system call filtering. Static filtering makes filter creation and usage more accessible via libraries (`libseccomp` [104]), automation (`sysfilter` [57]), or architecture portability (ABHAYA [168]). Dynamic filtering improves soundness of system call filters using automated testing [105, 223] or dynamic profiling (Confine [79]). Some approaches leverage a temporal context [2, 80, 133] enabling specific system calls during distinct execution phases. These approaches

are all coarse-grained and rely on whitelisting as a primary means of defense. BASTION goes beyond specifying a whitelist of allowed system calls, even if that whitelisting is made dynamic as in Temporal Filtering [80]. Notably, there exist several CVE's (e.g., Control Jujutsu [68], AOCR [185]) capable of bypassing Temporal Filtering as these attacks leverage system calls still permitted in the serving phase of an application. For this reason, BASTION is significantly more secure than any coarse-grained system call filtering approach.

Data integrity. Data integrity seeks to protect data such that no data in a program is corruptible either by tracing or employing a secure data copy. Specialized data integrity narrows coverage to protect control-dependent data (OAT [206]), developer annotated data (Datashield [37]), or data passed as arguments in callsites (Saffire [150]). Compared to these data integrity mechanisms, BASTION only enforces argument integrity for *sensitive variables*.

4.13 Summary

This work is built on the insight that regardless of attack complexity or end goal, a majority of attacks *must* leverage system calls to complete their attack. We believe BASTION makes important contributions in designing practical defense mechanisms with low performance overhead and high coverage. Instead of performing fine-grained integrity enforcement (e.g., fine-grained CFI), to increase coverage, BASTION focuses on protecting sensitive system call invocations, which are essential in attack chains, more comprehensively compared to other defenses. Thus, we presented three specialized system call contexts (Call-Type, Control-Flow, & Argument Integrity) for securing their legitimate usage and implement them with our prototype, BASTION. Evaluating the performance impact of BASTION using system call-intensive real-world applications, we demonstrate a low runtime overhead. This shows BASTION is a practical defense on its own to block an entire attack class that leverages system calls.

Application	NGINX (32 workers)	SQLite	vsFTPd
`execve`	0	0	0
`execveat`	0	0	0
`fork`	0	0	0
`vfork`	0	0	0
`clone`	96	48	36
`ptrace`	0	0	0
`mprotect`	334	501	7
`mmap`	534	42	33
`mremap`	0	0	0
`remap_file_pages`	0	0	0
`chmod`	0	0	0
`setuid`	32	0	12
`setgid`	32	0	12
`setreuid`	0	0	0
`socket`	32	1	85
`connect`	32	0	8
`bind`	1	1	77
`listen`	2	1	77
`accept`	0	11	87
`accept4`	5,665	0	0
Total BASTION monitor hook	6,713	557	433

Table 4.4: Sensitive system call usage from benchmarking.

Application	**NGINX**	**SQLite**	**vsftpd**
Total # application callsites	7,017	12,253	4,695
Total # arbitrary direct callsites	6,692	12,026	4,688
Total # arbitrary in-direct callsites	325	227	7
Total # sensitive callsites	26	13	12
Total # sensitive system calls called indirectly	0	0	0
`ctx_write_mem()`	5,226	1,337	204
`ctx_bind_mem()`	43	18	33
`ctx_bind_const()`	18	13	9
Total instrumentation sites	5,287	1,368	246

Table 4.5: Instrumentation statistics for BASTION.

Attack Category & Type	Violated Context CT	CF	AI
Return-oriented programming (ROP)			
Execute user command [12, 15, 23, 33, 66, 127, 144, 186, 187, 188, 189, 197, 216]	×	✓	✓
Execute root command [197]	×	✓	✓
Alter memory permission [17, 60, 69, 234]	×	✓	✓
Direct system call manipulation			
NEWTON CsCFI Attack [222]	✓	✓	✓
AOCR NGINX Attack 1 [185]	✓	✓	✓
CVE-2016-10190 ffmpeg [158] CVE-2016-10191 ffmpeg [159] CVE-2015-8617 php [157] CVE-2012-0809 sudo [153] CVE-2013-2028 nginx [154] CVE-2014-8668 libtiff v4.0.6 [156] CVE-2014-1912 python-2.7.6 [155]	✓	✓	✓
Indirect system call manipulation			
NEWTON CPI Attack [222]	✓	✓	✓
AOCR Apache Attack [222]	×	✓	✓
AOCR NGINX Attack 2 [185]	×	×	✓
COOP Against Google Chrome [49]	×	×	✓
Control Jujutsu against NGINX [68]	×	×	✓

CT: Call Type CF: Control Flow AI: Argument Integrity

Table 4.6: Real-world and synthesized exploits blocked by BASTION. ✓ denotes that a context can block an exploit, and × indicates that an exploit can bypass the context.

BASTION Configuration	Runtime & % Overhead Added Per Checkpoint NGINX	SQLite	vsftpd
BASTION + `file system syscalls` (seccomp hook only)	110.41 (0.15%)	36,993.27 (0.29%)	10.76 (0.08%)
BASTION + `file system syscalls` (fetch process state)	4.56 (95.88%)	7,461.18 (79.89%)	10.95 (1.85%)
BASTION + `file system syscalls` (full context checking)	3.65 (96.70%)	7,419.50 (80.00%)	11.01 (2.41%)

Table 4.7: BASTION Performance overhead when file system-related system calls (*e.g.*, `open`, `read`, `write`, `send`, `recv`) and variants (*e.g.*, `openat`, `sendfile`) are protected. The baseline is the unprotected version. In this scenario, fetching process state using `ptrace` contributes the most performance overhead.

Chapter 5

Future Work

In this chapter, future research directions are described for this dissertation. Both MARDU and BASTION exploit mitigation designs have fully working prototype implementations, but still have many possible extensions that could be further explored and investigated. The following discussion highlights and briefly explains several future research directions for MARDU and BASTION.

5.1 MARDU

- **Support for additional re-randomization triggers.** The implementation of MARDU presented in this dissertation only has one re-randomization trigger encoded. Specifically, MARDU currently activates re-randomization when a thread or process crash is registered by the OS kernel. However, security literature [67, 76, 85, 122, 165] reports that there are rare instances where code re-use does not need to crash a process or thread in order to disclose the victim program's code memory layout or corrupt program data (*e.g.*, crash-resistant probing). For this reason, MARDU should be extended to support other re-randomization triggers and strengthen its ability to proactively protect the host system. It would be beneficial to deduce other attacker-originating runtime side effects that could be leveraged to accurately detect when a system is under threat. Otherwise, seemingly benign attacker-induced side effects could be overlooked by mistake. Nevertheless, even if attacker side-effects are infered, the core challenge of this future direction resides in determining legitimate runtime behavior from one that is attacker-induced in a swift manner.

- **Support for protection of *legacy* binary executables.** While the majority of modern applications have source code available that can be recompiled to support MARDU, there exist a subset of applications for whom source code is unavailable. This can occur in cases where applications contain proprietary information or are potentially "legacy" and only exist in binary form. Since MARDU is implementated to leverage a compiler framework in order derive metadata and perform necessary instrumentation, MARDU is currently unable to be applied to programs without available source code.

  To get around this difficulty, the computing community traditionally resorts to binary rewriting [167]. The process of binary rewriting modifies a given compiled and possibly (dynamically) linked program in a way that it remains executable without access to the source code for recompilation [131]. Binary rewriting [230] is capable of directly modifying existing executables to achieve various goals such as observation [160], emulation [19], optimization [28, 58], and even security hardening [88, 235]. By leveraging binary re-writing, MARDU could per-

form structural recovery as well as label and symbol extraction of the program in order to then perform static binary transformation that is equivalent to MARDU's compiler instrumentation. This would then make an executable follow MARDU control-tranfer semantics, and enable the executable to be capable of runtime re-randomization.

- **Extend the x86-64 architecture to implement MARDU in hardware.** The MARDU's current runtime re-randomization mechanism is achieved with a compiler and kernel OS co-design that is considered a *system software* solution. In particular, MARDU implements a new control-flow transfer mechanism leveraging the concept of trampolines to move between the separate code and trampoline regions to protect forward edges, while employing a shadow stack to protect backward edges. This essentially imposes a completely different calling convention to accommodate live thread migration of randomized code without stopping computation. Moreover, MARDU utilizes memory protection keys (MPK) in order to enforce execute-only memory and protect these memory regions from attack. As a result, MARDU cannot guarantee its security protections in edge cases where applications contain and use `wrpkru` instructions. Additionally, while MARDU's current performance evaluation is quite reasonable, its performance could be significantly improved with re-implementation in hardware.

  Parts of the security community recognize the challenges and drawbacks associated with software solutions and instead of turned to designing and implementing solutions in hardware. Works such as Morpheus [74], CHERI [229], and HardBound [59] demonstrate that this is a feasible research direction to pursue. Some of MARDU's key features that would benefit from a hardware implementation would foremost be the trampoline control-flow transfer mechanism, the unique memory code and trampoline regions, and the shadow stack. This could all be done using gem5 [24], a popular open-source system-level and processor simulator in computer system architecture.

5.2 BASTION

- **Further reducing BASTION overhead to broaden the syscall protection scope.** The current BASTION prototype presented in this dissertation leverages a compiler component together with a runtime monitor to implement this defense system. While the compiler incurs negligible performance overhead (argument integrity context adds minimal instrumentation), the implementation of the monitor could be greatly improved. Recall that the runtime monitor is deployed as an separate process that is paired with the application to be protected by BASTION. In this regard, the necessity of an additional process limits the scalability of applying BASTION system-wide. Moreover, having the monitor deployed as a separate process requires costly inter-process communication (IPC) to be employed in order to relay runtime state information from the target program back to the monitor in order to perform verification of the call-type, control-flow, and argument integrity contexts. BASTION currently leverages the system call `process_vm_readv()` as its inter-process communication mechanism, rather than rely on similar but more expensive IPC mechanisms, such as Message Passing Interface (MPI).

Therefore, to further improve BASTION, particularly its runtime monitor, one viable solution could be to re-implement the runtime verification system as a Linux kernel module. By being implemented as a kernel module, BASTION can immediately gain access to resources and privileges that are otherwise only available to the kernel. More specifically, BASTION could recover the runtime state information (*e.g.*, general purpose register contents, stack) of a BASTION protected application transparently and significantly faster as well as efficiently reserve memory for its secure hash table. This would directly reduce performance degradation incurred by using BASTION and also make BASTION more scalable. Even more importantly, BASTION would be able to leverage the inherent user/kernel separation to completely isolate itself from an attacker.

- **Support for protection of other security-critical functions.** So far, the current BASTION prototype supports enforcement of system call integrity for over 20 security-critical system calls. BASTION could be extended to support other security-sensitive function invocations and broaden the reach of BASTION's specialized system call filtering mechanism. From a brief security study performed as BASTION was being designed, it was ascertained that two additional function types should covered by BASTION to provide an stronger defense mechanism.

  First, system calls who fall under the category of enabling information disclosure should be protected. To elaborate, this includes system calls that perform write operations such as `write`, `sendfile64`, `sendto`, `sendmsg`, and `sendmmsg`. An informal code study was done to learn more about the usage of system calls in modern code bases (*e.g.*, how often they appear in code, how often they are invoked during runtime, what types of arguments they are passed, etc.). It was discovered that information disclosure system calls are frequently repeatedly invoked throughout execution, exceptionally moreso than other security-critical system calls. While information disclosure system calls cannot directly hijack control of a victim system, they should still be considered dangerous; instead, information disclosure system calls can be a powerful tool for attackers enabling them to maliciously corrupt memory or relay memory layout information to an external destination.

  Second, security-sensitive library functions, especially functions such as `system()` and `dlopen()` should be protected. These library calls should considered security-critical foremost because they are ubiquitous by being part of the standard C library. Particularly, the `system()` library function allows for arbitrary computation by creating a child process that executes the shell command specified by a `char*` string input argument; the `dlopen()` function is responsible for loading a dynamic shared object (*e.g.*, a shared library) into process memory specified by a `char*` null-terminated string filename argument. As demonstrated in security literature [36, 67], pointers are especially vulnerable to memory corruption. As such, these two function's arguments could be potentially corrupted and subsequently leveraged to aid in completion of a code re-use attack. Note that BASTION is currently implemented to hook on invocations of security critical *system calls*; in Linux, a system call invocation can easily be disciphered in assembly code as the `syscall` instruction (but the legacy way of invoking syscalls via the software interrupt `int 0x80` still comes up occasionally). BASTION relied on leveraging the uniqueness of this system call calling convention in order to minimize the interference imposed by its runtime monitor. On the other hand, *ordinary library function calls* such as `system()` and `dlopen()` follow a completely different calling convention (*e.g.*,

relying on conventional `call` or `jmp` instructions).

Chapter 6

Conclusion

This dissertation presents a new perspective for designing practical exploit mitigations by securing distinct vital features leveraged by code re-use attacks. Formalizing the process of code re-use attacks into clear-cut steps greatly facilitated the process to identify which code components are leveraged in code re-use, and therefore *security critical*, as well as additionally aiding to identify prevalent processes across attack vectors. Code components are the foundational building blocks that are malicously corrupted and used in various code re-use attack vectors. This dissertation explores these *security critical* code components in order to define the bounds for the scope of this dissertation. The derivation of security critical code components in this dissertation was pivotal towards creation of focused defense systems; rather than designing a system that has broad but shallow coverage of application code, this dissertation concentrates its work efforts to identifying quintessential cruxes in code re-use and hardening those code components in a robust, yet light-weight, manner. In doing so, this dissertation advanced the current understanding of what code components and processes are absolutely crucial to protect without fail in production code.

This dissertation recognizes the lack of adoption of strong defense techniques in current mainstream computing because of unrealistic performance overheads or insufficent scalability, for the security guarantees provided. Consequently, current state-of-the-art defense mechanisms relevant to code re-use were studied to identify clear weaknesses and discern potential points of improvements. The formalization of the process of code re-use attacks was extended to also map the complimentary defense strategies to each step within. This compendium of code re-use versus current state-of-the-art defense strategies allowed for identification of promising directions to pursue in this dissertation and contribute to answering the question of how to design strong and realistically practical defense systems for modern day computing.

This dissertation first describes the investigation of code re-use attack's relationship with code gadgets and how randomization can inhibit completion of code gadget chains. With the preliminary understanding that applying randomization to a process's memory layout is a valid defense strategy, this dissertation designs a novel randomization defense system, MARDU. MARDU is unique compared to both static and active randomization schemes in order to compensate for each approach's weaknesses, respectively. Static randomization is performed as a one-shot effort, as such, a single memory disclosure during runtime can entirely invalidate this defense strategy. Active randomization, on the other hand, makes the creation of gadget chains significantly more difficult as now code is a constantly moving target within the system. At the same time, the active randomization scheme introduces many moving pieces and is therefore much more complex, to the point of unintentionally slowing down the rate of computation, even when not under siege from an attacker. Moreover, active randomization makes sharing of code, especially API libraries, extremely difficult since randomization breaks the assumptions of an API interface having function entries at known

static locations. In order to enable code-sharing some level of consistency must be maintained across the entire system.

This dissertation describes the complete design, implementation, and evaluation of MARDU, an on-demand, coding-sharing-capable, randomization defense system. MARDU achieves scalable code-sharing of diversified code from applications and shared libraries by employing the concept of code trampolines to introduce a level of indirection to separate code function entries from code function bodies together with randomization. Compiler analysis and transformation passes are used in order to integrate and support code trampolines with traditional control-flow transfer semantics; the Linux kernel process memory management component is adjustmented to support a new layout and intentionally isolate code from trampolines to maintain security. This design removed redundant computation like tracking, and patching and inherently achieved deduplication of code, such that code can be shared transparently and securely across the system without any additional consistency mechanisms in place. MARDU runtime re-randomization mechanism is designed to only activate when attacker activities, such as a thread unexpectedly crashing, are detected on the host system. The benefits of this on-demand approach is twofold. By randomizing code only when attacker activities are detected, code and code pointer leakage can be prevented. Second, the host nor the target program do not have to be burdened to carry out unnecessary randomization compared to active randomization.

MARDU has been evaluated with both standard performance benchmarks (SPEC CPU2006) as well as real-world applications such as the NGINX webserver to give a realistic assessment of how MARDU would behave being deployed in a production environment. MARDU's performance degradation was incurred no more than 5.5% and 4.4%, respectively. Even when under constant (synthetic) attack, MARDU only adds no more than 15% additional overhead for the most complex applications. Security-wise MARDU successfully defended against all attacks relevant to undermining randomization defense mechanisms including JIT-ROP attacks, code inference attacks, low-profile timing attacks, and code pointer offsetting attacks.

This dissertation then pursued the implications of completely securing system calls in order to thwart code re-use attacks. Reviewing literature concerning code re-use attack variants and respective defense strategies revealed that leveraging system calls are a shared attack step for many prominent attack vectors, and therefore a valuable defense objective. Moreover, a survey of open-source production code bases revealed that system calls are used very sparingly and thus have significantly less instances that need protection compared to other code components commonly sought by attackers. From a survey of literature regarding system call protection, a similar finding of trends was observed as in the case of randomization defenses. For those defense strategies who were light-weight, they were often fragile or offered a coarse-grained level of protection for system calls, not nearly enforcing tight enough constraints to prevent arbitrary execution. For defense strategies that employed context-sensitive approaches, protecting aspects such as data integrity, they largely incurred excessive performance overhead to be practical for production deployment. As a result, there was a clear opportunity to design a defensive strategy that could apply strong context-sensitive approaches in a very narrow scope having identified system calls as a critical lynchpin in code re-use attack completion.

This dissertation uses this insight regarding system calls proposes the concept of *System Call Integrity*, a novel security policy that enforces the correct use of system calls in a program. In order

to enforce System Call Integrity, this dissertation develops three novel system call contexts that are sufficient enforce legitimate system call usage during runtime. The first context *Call-Type* ensures system calls that are actually used by an application are always called with the expected calling convention using in code: either with an direct or indirect call. The second context *Control-Flow* ensures control-flow paths reaching a system call are legitimate by assessing a narrower form of CFI. By employing this narrower form of CFI, implementation of and runtime verification this context will interfere significantly less compared to traditional CFI which blankets the entire application. Finally, the third context *Argument Integrity* verifies value integrity for all system call arguments and all data-dependent variables such that all system call arguments are genuine. Traditionally, applying data integrity application wide is very performance intensive; by applying it to a subset of critical data, it is considerably more feasible to reach practical performance overhead.

This dissertation describes the complete design, implementation, and evaluation of BASTION, a context-sensitve, system call filtering defense system built around the three system call contexts that allow for the enforcement of system call integrity. BASTION protects a subset of *security sensitive* system calls (20) rather than the full system call list, which currently numbers several hundred. BASTION leverages compiler analysis and instrumentation passes in order derive the necessary metadata to enforce each of the three contexts at runtime; a separate monitor process is connected to each protected application and is responsible for verifying all three contexts. Instrumentation is only necessary for the *Argument Integrity* context, which writes a secure copy of argument values to a secure hash table whenever they are updated in the code. In order to optimize performance, runtime verification only occurs before a sensitive system call invocation. BASTION leverages the sparse nature of system call invocations in order perform a single comprehensive check at an invocation point rather than constantly impede progress of computation at each control-flow transfer. Coincidently, by solely ensuring that system calls are adequately protected, all attack classes depending on system call usage can be effectively nullified as well.

BASTION has been evaluated with a multitude of case studies for its security evaluation and paired with three real-world applicaions, a webserver, a database engine, and an FTP server to investigate its performance implications. Even with all three context checks and CET enabled, BASTION's performance overhead was 2.01% in the worst case, when deployed with the SQLite database engine, and negligible at best, when deployed with the NGINX web server. BASTION successfully defeated all 32 case-studies borrowed from other prominent security literature that included both real-world vulnerabilities as well as synthesized attacks. This security evaluation demonstrated the strength of the unified front which the three system call contexts provide when used together.

In closing, this dissertation advances pragmatic exploit mitigation design against code re-use attacks. The study of the code re-use attack process and corresponding defense strategies in this dissertation allowed for identification of their respective core features, such that the design and implementation of two new separate practical defense systems have been demonstrated. MARDU addressed the lack of code sharing and wasted CPU cycles in continuous randomization defenses while BASTION identified and addressed system calls as a security critical lynchpin in attacks that must be protected to the greatest extent possible.

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[2] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. Shard: Fine-grained kernel specialization with context-aware hardening. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[3] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: Debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 70–83, 2019.

[4] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Junaid Haroon Siddiqui, and Fareed M Zaffar. Trimmer: An automated system for configuration-based software debloating. *IEEE Transactions on Software Engineering*, 2021.

[5] Salman Ahmed, Ya Xiao, Kevin Z Snow, Gang Tan, Fabian Monrose, and Danfeng Yao. Methodologies for quantifying (re-) randomization security and timing under jit-rop. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1803–1820, 2020.

[6] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers: principles, techniques, and tools addison. *Wesly Publishing Company, Reading, Massachusetts*, 2:502–503, 1986.

[7] One Aleph. Smashing the stack for fun and profit. *http://www. shmoo. com/phrack/Phrack49/p49-14*, 1996.

[8] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. Lightweight, multi-stage, compiler-assisted application specialization. *arXiv preprint arXiv:2109.02775*, 2021.

[9] Amazon. Amazon EC2 C5 Instances, 2019. https://aws.amazon.com/ec2/instance-types/c5/.

[10] Autore Anonimo. Once upon a free ().. *Phrack Magazine*, 11(57), 2001.

[11] ARM Community. What is eXecute-Only-Memory (XOM)?, july 2017. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/what-is-execute-only-memory-xom.

[12] Miguel A. Arroyo. Simple ROP Exploit Example, 2022. https://gist.github.com/mayanez/c6bb9f2a26fa75261a9a26a0a637531b/.

[13] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[14] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, November 2014.

[15] Andrew Bae. Bypass DEP/NX and ASLR with Return Oriented Programming Technique, 2019. https://medium.com/4ndr3w/linux-x86-bypass-dep-nx-and-aslr-with-return-oriented-programming-ef4768363c9a/.

[16] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, 2014.

[17] Antonio Barresi. Bypassing non-executable memory, ASLR and stack canaries on x86-64 Linux, 2014. https://www.antoniobarresi.com/security/exploitdev/2014/05/03/64bitexploitation/.

[18] Markus Bauer and Christian Rossow. Cali: Compiler-assisted library isolation. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 550–564, 2021.

[19] Fabrice Bellard. QEMU, a fast and portable dynamic translator. *USENIX annual technical conference, FREENIX Track*, 41(46):10–5555, 2005.

[20] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[21] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V Mancini. Operating system enhancements to prevent the misuse of system calls. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 174–183, 2000.

[22] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[23] Patrik Billgren. Analysis of Defenses against Return Oriented Programming. Master's thesis, Lund University, 2014. https://www.eit.lth.se/sprapport.php?uid=829/.

[24] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[25] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[26] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, page 30–40, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305648. doi: 10.1145/1966913.1966919. URL https://doi.org/10.1145/1966913.1966919.

[27] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[28] Bryan Buck and Jeffrey K Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[29] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing {PHP} applications with tailored system call allowlists. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2881–2898, 2021.

[30] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[31] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[32] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[33] c0ntex. Bypassing non-executable-stack during Exploitation (return-to-libc), 2006. https://www.exploit-db.com/papers/13204/.

[34] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop*, pages 139–151, 2021.

[35] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[36] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.

[37] Scott A Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 193–204, 2017.

[38] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[39] Checkoway, Stephen and Davi, Lucas and Dmitrienko, Alexandra and Sadeghi, Ahmad-Reza and Shacham, Hovav and Winandy, Marcel. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, October 2010.

[40] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[41] Xi Chen, Herbert Bos, and Cristiano Giuffrida. CodeArmor: Virtualizing The Code Space to Counter Disclosure Attacks. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (Euro S&P)*, Paris, France, April 2017.

[42] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. Ropecker: A generic and practical approach for defending against rop attack. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014.

[43] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Vik: practical mitigation of temporal memory safety violations through object id inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–284, 2022.

[44] Chris Evans. vsftpd, 2022. https://security.appspot.com/vsftpd.html.

[45] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies*, 2016(4):454–469, 2016.

[46] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[47] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Return to where? you can't exploit what you can't find. *Proceedings of Black Hat USA*, 2015.

[48] Stephen Crane, Andrei Homescu, and Per Larsen. Code Randomization: Haven't We Solved This Problem Yet? In *Cybersecurity Development (SecDev), IEEE*, pages 124–129. IEEE, 2016.

[49] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRaP: Table Randomization and Protection Against Function-reuse Attacks. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[50] Charlie Curtsinger and Emery D Berger. Stabilizer: Statistically sound performance evaluation. *ACM SIGARCH Computer Architecture News*, 41(1):219–228, 2013.

[51] D. Richard Hipp. SQLite, 2022. https://www.sqlite.org/index.html.

[52] Dan Kegel. dkftpbench v0.45, 2005. http://www.kegel.com/dkftpbench/.

[53] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Singapore, Republic of Singapore, April 2015.

[54] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[55] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[56] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[57] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. `sysfilter`: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 459–474, 2020.

[58] DynInst Developers. Dyninst - dynamic instrumentation framework, 2016. https://dyninst.org/.

[59] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[60] dimi. ARM exploitation - Defeating DEP - executing mprotect, 2022. https://blog.3or.de/arm-exploitation-defeating-dep-executing-mprotect.html.

[61] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.

[62] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[63] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 478–493, 2019.

[64] Ulrich Drepper. How to write shared libraries. *Retrieved Jul*, 16:2009, 2006.

[65] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. HeapHopper: Brining Bounded Model Checking to Heap Implementation Security. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[66] Saif El-Sherei. Return-Oriented-Programming (ROP FTW), 2022. http://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf.

[67] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[68] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, page 901–913, Denver, Colorado, October 2015.

[69] exploit. 64-bit ROP | You rule 'em all!, 2022. https://0x00sec.org/t/64-bit-rop-you-rule-em-all/1937.

[70] F5, Inc. NGINX Web Server, 2022. https://nginx.org/en/.

[71] Fedora. Hardening Flags Updates for Fedora 28, 2018. https://fedoraproject.org/wiki/Changes/HardeningFlags28.

[72] Rich Felker. musl libc, 2022. https://musl.libc.org/.

[73] Lang Feng, Jiayi Huang, Jeff Huang, and Jiang Hu. Toward taming the overhead monster for data-flow integrity. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(3):1–24, 2021.

[74] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 469–484, Providence, RI, USA, April 2019.

[75] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[76] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[77] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[78] Masoud Ghaffarinia and Kevin W Hamlen. Binary control-flow trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1009–1022, 2019.

[79] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 443–458, 2020.

[80] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1749–1766, 2020.

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

[82] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, August 2012.

[83] Will Glozer. a HTTP benchmarking tool, 2019. https://github.com/wg/wrk.

[84] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[85] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining Information Hiding (and What to Do about It). In *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, August 2016.

[86] Jens Grossklags and Claudia Eckert. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Proceedings of the 21th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Heraklion, Crete, Greece, September 2018.

[87] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, Scottsdale, AZ, March 2017.

[88] William H Hawkins, Jason D Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W Davidson. Zipr: Efficient static binary rewriting for security. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 559–566. IEEE, 2017.

[89] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-process Isolation for High-throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[90] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, pages 380–394, Toronto, ON, Canada, October 2018.

[91] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd My Gadgets Go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[92] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. librando: Transparent Code Randomization for Just-in-Time Compilers. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 993–1004, Berlin, Germany, October 2013.

[93] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of {Data-Oriented} exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, 2015.

[94] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[95] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[96] Intel Corporation. Control-flow Enforcement Technology Specification, may 2019. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[97] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2019. https://software.intel.com/en-us/articles/intel-sdm.

[98] Intel Corporation. INTEL ® XEON ® SCALABLE PROCESSORS, 2019. https://www.intel.com/content/www/us/en/products/processors/xeon/scalable.html.

[99] Intel Corporation. Intel Memory Protection Extensions (Intel MPX), 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[100] Intel Corporation. Intel PT, June 2022. https://perf.wiki.kernel.org/index.php/Perf_tools_support_for_Intel%C2%AE_Processor_Trace.

[101] Intel Corporation. Intel Transactional Memory (Intel TSX), 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[102] Mohannad Ismail, Jinwoo Yom, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. VIP: safeguard value invariant property for thwarting critical memory corruption attacks. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 1612–1626. ACM, 2021. doi: 10.1145/3460120.3485376.

[103] Ismail, Mohannad and Yom, Jinwoo and Jelesnianski, Christopher and Jang, Yeongjin and Min, Changwoo. VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1612–1626, 2021.

[104] Jake Edge. A library for seccomp filters, April 2012. https://lwn.net/Articles/494252/.

[105] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 37–48, 2016.

[106] Sunwoo Jang, Somin Song, Byungchul Tak, Sahil Suneja, Michael V. Le, Chuan Yue, and Dan Williams. Secquant: Quantifying container system call exposure. In *Proceedings of the 27th European Symposium on Research in Computer Security (ESORICS)*, page 145–166, 2022.

[107] Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang. Mardu: Efficient and scalable code re-randomization. In *Proceedings of the 13th ACM International Systems and Storage Conference*, pages 49–60, 2020.

[108] Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang. MARDU: efficient and scalable code re-randomization. In *SYSTOR 2020: The 13th ACM International Systems and Storage Conference, Haifa, Israel, October 13-15, 2020*, pages 49–60. ACM, 2020.

[109] Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang. Securely sharing randomized code that flies. *Digital Threats: Research and Practice*, 2022.

[110] Jonathan Corbet. x86 NX support, 2004. https://lwn.net/Articles/87814/.

[111] Jonathan Corbet. Securely renting out your CPU with Linux, January 2005. https://lwn.net/Articles/120647/.

[112] Jonathan Corbet. New system calls for memory management, May 2019. https://lwn.net/Articles/789153/.

[113] Michel Kaempf. Vudo malloc tricks. phrack magazine, 57 (8), august 2001, 2001.

[114] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, October 2003.

[115] The kernel development community. Seccomp BPF (SECure COMPuting with filters), 2015. https://lwn.net/Articles/656307/.

[116] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive call-site sensitive control flow integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 95–110. IEEE, 2019.

[117] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive call-site sensitive control flow integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 95–110. IEEE, 2019.

[118] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 195–211, 2019.

[119] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2006.

[120] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[121] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[122] Benjamin Kollenda, Enes Göktaş, Tim Blazytko, Philipp Koppe, Robert Gawlik, Radhesh Krishnan Konoth, Cristiano Giuffrida, Herbert Bos, and Thorsten Holz. Towards Automated Discovery of Crash-resistant Primitives in Binary Executables. In *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN)*, Denver, CO, June 2017.

[123] Hyungjoon Koo and Michalis Polychronakis. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–June 2016.

[124] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[125] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, pages 1–6, 2019.

[126] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163, Broomfield, Colorado, October 2014.

[127] NYU OSIRIS Lab. ROP-CTF101, 2022. https://ctf101.org/binary-exploitation/return-oriented-programming/.

[128] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[129] Larabel, Michael. Glibc 2.28 Released With Unicode 11.0 Support, Statx & Intel Improvements, 2018. https://www.phoronix.com/news/Glibc-2.28-Released.

[130] Larabel, Michael. Intel Confirms CET Security Support For Tiger Lake, 2020. https://www.phoronix.com/news/Intel-CET-Tiger-Lake.

[131] James R Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 291–300, 1995.

[132] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[133] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. Speaker: Split-phase execution of application containers. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 230–251. Springer, 2017.

[134] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. Fine-cfi: fine-grained control-flow integrity for operating system kernels. *IEEE Transactions on Information Forensics and Security*, 13(6):1535–1550, 2018.

[135] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1835, 2020.

[136] Linux Programmer's Manual. ELF(5) – Linux manual page, 2022. https://man7.org/linux/man-pages/man5/elf.5.html.

[137] Linux Programmer's Manual. PTRACE(2) – Linux manual page, 2022. https://man7.org/linux/man-pages/man2/ptrace.2.html.

[138] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[139] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and efficient CFI enforcement with intel processor trace. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX, February 2017.

[140] Kangjie Lu and Hong 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*, pages 1867–1881, 2019.

[141] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[142] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[143] lwn.net. GNU C Library 2.28 released, 2018. https://lwn.net/Articles/761462/.

[144] Ben Lynn. 64-bit Linux Return Oriented Programming, 2022. https://crypto.stanford.edu/~blynn/rop/.

[145] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[146] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016.

[147] Michael Eager. The DWARF Debugging Standard, 2021. https://dwarfstd.org/.

[148] Microsoft Support. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003, 2017. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in.

[149] Shachee Mishra and Michalis Polychronakis. Shredder: Breaking Exploits Through API Specialization. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 1–16, 2018.

[150] Shachee Mishra and Michalis Polychronakis. Saffire: Context-sensitive function specialization against code reuse attacks. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 17–33. IEEE, 2020.

[151] Maliheh Monshizadeh, Prasad Naldurg, and VN Venkatakrishnan. Mace: Detecting privilege escalation vulnerabilities in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 690–701, 2014.

[152] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[153] National Institute of Standards and Technology. National Vulnerability Database. CVE-2012-0809, 2015. https://nvd.nist.gov/vuln/detail/CVE-2012-0809.

[154] National Institute of Standards and Technology. National Vulnerability Database. CVE-2013-2028, 2015. https://nvd.nist.gov/vuln/detail/CVE-2013-2028.

[155] National Institute of Standards and Technology. National Vulnerability Database. CVE-2014-1912, 2015. https://nvd.nist.gov/vuln/detail/CVE-2014-1912.

[156] National Institute of Standards and Technology. National Vulnerability Database. CVE-2014-8668, 2015. https://nvd.nist.gov/vuln/detail/CVE-2014-8668.

[157] National Institute of Standards and Technology. National Vulnerability Database. CVE-2015-8617, 2015. https://nvd.nist.gov/vuln/detail/CVE-2015-8617.

[158] National Institute of Standards and Technology. National Vulnerability Database. CVE-2016-10190, 2016. https://nvd.nist.gov/vuln/detail/CVE-2016-10190.

[159] National Institute of Standards and Technology. National Vulnerability Database. CVE-2016-10191, 2016. https://nvd.nist.gov/vuln/detail/CVE-2016-10191.

[160] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42 Issue 6, pages 89–100. ACM, 2007.

[161] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 577–587, Edinburgh, UK, June 2014.

[162] Ben Niu and Gang Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317–1328, 2014.

[163] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[164] NuoDB, Inc. DBT-2, 2022. https://github.com/nuodb/dbt2.

[165] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking holes in information hiding. In *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, August 2016.

[166] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–30, 2018.

[167] Pádraig O'sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. Retrofitting security in cots software with binary rewriting. In *Ifip International Information Security Conference*, pages 154–172. Springer, 2011.

[168] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 1–26, 2020.

[169] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[170] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[171] Pardis Pashakhanloo, Aravind Machiry, Hyonyoung Choi, Anthony Canino, Kihong Heo, Insup Lee, and Mayur Naik. Pacjam: Securing dependencies continuously via package-oriented debloating. In *Proceedings of the 17th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Nagasaki, Japan, May 2022.

[172] Mathias Payer, Antonio Barresi, and Thomas R Gross. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.

[173] Jannik Pewny, Philipp Koppe, Lucas Davi, and Thorsten Holz. Breaking and Fixing Destructive Code Read Defenses. In *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 55–67, Abu Dhabi, UAE, April 2017.

[174] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. kR^ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[175] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–180, 2020.

[176] Chris Porter, Sharjeel Khan, and Santosh Pande. On-the-fly code activation for attack surface reduction. *arXiv preprint arXiv:2110.09557*, 2021.

[177] Chris Porter, Sharjeel Khan, and Santosh Pande. On-the-fly code activation for attack surface reduction. *arXiv preprint arXiv:2110.09557*, 2021.

[178] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[179] Soumyakant Priyadarshan, Huan Nguyen, and R Sekar. Practical fine-grained binary code randomization. In *Annual Computer Security Applications Conference*, pages 401–414, 2020.

[180] Chenxiong Qian, Hong Hu, Mansour A Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[181] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium (Security)*, pages 869–886, Baltimore, MD, August 2018.

[182] Qualcomm, Inc. Pointer Authentication on ARMv8.3, january 2017. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[183] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. BinTrimmer: Towards static binary debloating through abstract interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 482–501. Springer, 2019.

[184] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[185] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[186] Juan Sacco. PMS 0.42 - Local Stack-Based Overflow (ROP), 2018. https://www.exploit-db.com/exploits/44426/.

[187] Juan Sacco. Crashmail 1.6 - Stack-Based Buffer Overflow (ROP), 2018. https://www.exploit-db.com/exploits/44331/.

[188] Jonathan Salwan. PHP 5.3.6 - Local Buffer Overflow (ROP), 2011. https://www.exploit-db.com/exploits/17486/.

[189] Jonathan Salwan. Return Oriented Programming and ROPgadget tool, 2022. http://shell-storm.org/blog/Return-Oriented-Programming-and-ROPgadget-tool/.

[190] Jonathan Salwan. ROPgadget - Gadgets finder and auto-roper, 2022. http://www.shell-storm.org/project/ROPgadget/.

[191] Sascha Schirra. Ropper - ROP gadget finder and binary information tool, 2022. https://github.com/sashs/Ropper.

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

[193] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, October–November 2007.

[194] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October–November 2007.

[195] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–11, 2019.

[196] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. Trimmer: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 329–339, 2018.

[197] shaun2k2. Exploitation - Returning to libc, 2006. https://www.exploit-db.com/papers/13197/.

[198] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. FreeGuard: A Faster Secure Heap Allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[199] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A Tunable Secure Allocator. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[200] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[201] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[202] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[203] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Pack. HDFI: Hardware-Assisted Data-flow Isolation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[204] sourceware.org. V2 [PATCH 24/24] Intel CET: Document –enable-cet, 2018. https://sourceware.org/legacy-ml/libc-alpha/2018-07/msg00550.html.

[205] Mingshen Sun, John CS Lui, and Yajin Zhou. Blender: Self-randomizing address space layout for android apps. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Evry, France, September 2016.

[206] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449. IEEE, 2020.

[207] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[208] The Clang Team. Clang 12 documentation - Control Flow Integrity, 2021. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[209] The Clang Team. Clang 16 documentation: CONTROL FLOW INTEGRITY, 2022. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[210] The PAX Team. Address Space Layout Randomization, 2003. https://pax.grsecurity.net/docs/aslr.txt.

[211] The Santa Cruz Operation. System V Application Binary Interface MIPS RISC Processor Supplement, 3rd Edition., 1996. https://refspecs.linuxfoundation.org/elf/mipsabi.pdf.

[212] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[213] Sami Tolvanen. Control Flow Integrity in the Android kernel, 2018. https://security.googleblog.com/2018/10/posted-by-sami-tolvanen-staff-software.html.

[214] Torvalds, Linus. syscall_64.tbl, November 2022. https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall$_6$4.tbl.

[215] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *International Workshop on Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[216] Unknown. Introduction to Return Oriented Programming (ROP), 2013. https://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html/.

[217] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 1221–1238, Santa Clara, CA, August 2019.

[218] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419, 2017.

[219] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[220] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940, 2015.

[221] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[222] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[223] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining sandboxes for linux containers. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 92–102. IEEE, 2017.

[224] Zhe Wang, Chenggang Wu, Jianjun Li, Yuanming Lai, Xiangyu Zhang, Wei-Chung Hsu, and Yueqiang Cheng. Reranz: A Light-weight Virtual Machine to Mitigate Memory Disclosure Attacks. In *Proceedings of the 13th International Conference on Virtual Execution Environments (VEE)*, Xi'an, China, April 2017.

[225] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. Safehidden: An efficient and secure information hiding technique using re-randomization. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[226] Zhilong Wang and Peng Liu. Position paper: Gpt conjecture: understanding the trade-offs between granularity, performance and timeliness in control-flow integrity. *Cybersecurity*, 4 (1):1–9, 2021.

[227] Bryan C Ward, Richard Skowyra, Chad Spensky, Jason Martin, and Hamed Okhravi. The Leakage-Resilience Dilemma. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*, Luxembourg, September 2019.

[228] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, October 2012.

[229] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[230] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 52(3): 1–37, 2019.

[231] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–June 2016.

[232] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.

[233] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–147, 2020.

[234] ZadYree. HT Editor 2.0.20 - Local Buffer Overflow (ROP), 2012. https://www.exploit-db.com/exploits/22683/.

[235] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[236] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[237] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. One size does not fit all: security hardening of mips embedded systems via static binary debloating for shared libraries. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 255–270, 2022.

[238] Mingwei Zhang and R Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[239] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 631–644, Providence, RI, USA, April 2019.

[240] Andreas Ziegler, Julian Geus, Bernhard Heinloth, Timo Hönig, and Daniel Lohmann. Honey, I shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–23, 2019.