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

Christopher S. Jelesnianski

Preliminary Exam

Doctor of Philosophy

in

Computer Engineering

Changwoo Min, Chair

Yeongjin Jang

Wenjie Xiong

Danfeng (Daphne) Yao

Haibo Zeng

April 13, 2022

Blacksburg, Virginia

Keywords: Exploit Mitigation, Practical Design, System Software, Code Re-use Attacks, System Calls

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

Christopher S. Jelesnianski

(ABSTRACT)

Practically is an important, yet overlooked quality in exploit mitigation design. While many defense techniques have been proposed, few have become mainstream and deployed in production, specifically for their lack of practicality. With the small amount of defenses deployed, mediocre security is common. This has serious consequences, as data breaches and identity theft is now a common occurance in society. Practical design is important both improve deployment and to avoid adverse effects such as performance degradation or memory monopolization. In order to be practical, exploit mitigations should be performant, robust, scalable, and guarantee high security. However, balancing all 4 of these important features is hard to achieve in practice.

Practical design must navigate several difficult challenges. Because modern attacks have become capable of leveraging many types of code components, defenses have correspondingly needed to protect more aspects of code to block these attacks. Moreover, to sufficiently protect code, defense techniques have become increasingly complex. Supporting such fine-grained and complex defense schemes comes with inherent disadvantages like significant hardware resource utilization that could be otherwise used for useful work. Complexity has made performance, security, and scalability all competing ideals in practical code design. Some defenses have implemented schemes with negligible performance impact, however they come at the cost of a weaker security guarantee. A practical defense would not sacrifice any aspect to maintain other desirable properties.

This thesis proposal describes two practical exploit mitigation designs. This thesis proposal first presents MARDU, a re-randomization approach that utilizes on-demand randomization to block code re-use attacks. To the best of my knowledge, MARDU is the first re-randomization technique capable of code sharing for re-randomized code system wide. Additionally, this thesis proposal presents on-going work, BASTION, a system call filtering approach whose objective is to significantly strengthen security surrounding system calls. BASTION proposes three new specialized contexts for the effective enforcement of legitimate

system call usage. Both exploit mitigations presented in this work focus on achieving practicality in their design and implementation such that they may become mainstream defense techniques.

This thesis proposal describes the design, implementation, and evaluation of MARDU. Evaluating MARDU shows that re-randomization overhead can be significantly reduced for practical deployment without sacrificing randomization entropy. Moreover, MARDU shows it is capable of defeating prominent code re-use variants with this practical design. For BASTION, this thesis proposal describes its design and presents a preliminary evaluation of a bare-bones prototype. These initial results show negigible performance impact thus motivating for the completion and implementation of this design.

This thesis proposal also explains post-preliminary exam work. The proposed work includes completion of the BASTION prototype which includes a custom compiler and runtime enforcement monitor. Additionally, a comprehensive security and performance evaluation is proposed to supplement BASTION and substantiate the merits of this design.

Contents

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 Motivation . . . 1

1.1.1 Attacks, How they Attack . . . 2

1.1.2 Defenses, How they Defend . . . 3

1.1.3 Challenges in Practical Defense Design . . . 4

1.1.4 Goals . . . 8

1.2 Contributions . . . 9

1.3 Summary of Post-Preliminary Exam Work . . . 11

1.4 Thesis Organization . . . 12

2 Background 14

2.1 Attack-Targeted Code Components . . . 15

2.2 Prevalent Attacks 17

2.3 Modern Defense Techniques 20

3 MARDU: On-Demand, Shared, and Scalable Code Re-randomization 24

3.1 Introduction 24

3.2 Code Layout (Re-)Randomization 27

3.2.1 Attacks against Load-time Randomization 29

3.2.2 Defeating A1/A2 via Continuous Re-randomization 30

3.2.3 Attacks against Continuous Re-randomization 31

3.3 Threat Model and Assumptions 32

3.4 MARDU Design 33

3.4.1 Overview 33

3.4.2 MARDU Compiler 38

3.4.3 MARDU Kernel 42

3.5 Implementation 49

3.5.1 MARDU Compiler 50

3.5.2 MARDU Kernel 50

3.5.3 Limitation of Our Prototype Implementation 51

3.6 Evaluation 51

3.6.1 Security Evaluation 52

3.6.2 Performance Evaluation . . . 57
3.6.3 Scalability Evaluation . . . 59
3.7 Discussion and Limitations . . . 66
3.8 Summary . . . 71

4 BASTION: Context Sensitive System Call Protection 73

4.1 Introduction . . . 73
4.2 Background . . . 77
4.2.1 System Call Usage in Attacks . . . 77
4.2.2 Current System Call Protection Mechanisms . . . 78
4.3 Contexts for System Call Integrity . . . 80
4.3.1 Call-Type Context . . . 80
4.3.2 Control-Flow Context . . . 81
4.3.3 Argument Integrity Context . . . 82
4.3.4 Real-World Code Examples . . . 83
4.4 Threat Model and Assumptions . . . 86
4.5 BASTION Design Overview . . . 86
4.6 BASTION Compiler . . . 87
4.6.1 Analysis for Call-Type Context . . . 87
4.6.2 Analysis for Control-Flow Context . . . 89

4.6.3 Analysis for Argument Integrity Context 90
4.7 BASTION Runtime Monitor 94
4.7.1 Initializing BASTION Monitor 94
4.7.2 Enforcing Call-Type Context 96
4.7.3 Enforcing Control-Flow Context 97
4.7.4 Enforcing Argument Integrity Context 97
4.8 Preliminary Evaluation 98
4.8.1 Evaluation Methodology 98
4.8.2 Performance Evaluation 99
4.9 Related Work 101
4.10 Summary 103

5 Ongoing Research & Future Work **104**

5.1 Completed BASTION Prototype 104
5.2 Comprehensive Security Study of BASTION 105
5.3 Comprehensive Performance Study of BASTION 105

6 Conclusions **107**

Bibliography **110**

List of Figures

1.1 Conventional attack procedure to accomplish a code re-use attack 3

3.1 Overview of MARDU 37

3.2 Illustrative example executing a MARDU-compiled function foo(), which calls a function bar() and then returns. 40

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). 44

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. 47

3.5 MARDU performance overhead breakdown for SPEC 57

3.6 Performance comparison of NGINX web server 57

3.7 Cold load-time randomization overhead 60

3.8 Runtime re-randomization latency 61

3.9 Overhead varying re-randomization frequency 62

3.10 File size increase with MARDU compilation 62

3.11 Top 25 shared libraries with their reference count on our idle Linux server ordered from most linked to least linked libraries. 67

3.12 Estimated runtime memory savings with shared memory MARDU approach for the top 25 most linked libraries on our idle Linux server. 68

3.13 CDF of shared library occurance on a idle Linux server. 68

3.14 CDF plot of estimated runtime memory savings with MARDU's shared memory approach. 69

4.1 Legitimate use of the execve system call in NGINX. 84

4.2 Snippet of NGINX code that can be compromised by an attacker to achieve Attack 2, to call the mprotect system call elsewhere in the code base using a vulnerable code pointer v[index].get_handler(). 85

4.3 The design overview of BASTION. At compile time, BASTION analyzes a program and generates the program's context metadata. For the call-type and control-flow contexts, BASTION generates static metadata information. For the argument integrity context, BASTION generates static metadata information to check static argument values (*e.g.*, constant) and instruments the program to track its dynamic argument values. At runtime, the BASTION monitor catches all invocations of sensitive system calls and verifies all three contexts of the system call by checking the generated metadata together with dynamic runtime information. 88

4.4 BASTION's program instrumentation for argument integrity. BASTION binds constant arguments to specified values (NULL, -1, 0) using ctx_bind_const_X. It manages the shadow copy of memory-backed arguments (gshm->size, prots, b2) using ctx_write_mem when assigned. It then binds memory-backed direct arguments using ctx_bind_mem_X before calling the mmap system call. The protection of memory-backed arguments is extended using a field-sensitive use-def analysis (size field of gshm, b2←flags) at an inter-procedural level. . . 91

4.5 Performance breakdown for each of BASTION's system call contexts. . . . 99

List of Tables

1.1 Table showing the corresponding Defense Design Archetypes for each attack step. 3

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 [136, 141] 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. 28

3.2 Breakdown of MARDU instrumentation 63

4.1 BASTION library API for argument integrity. BASTION manages the shadow copy of sensitive variables (ctx_write_mem) and binds memory-backed (direct or extended) arguments and constant arguments (ctx_bind_mem_X, ctx_bind_const_X) to a specific position (X-th argument) of a system call callsite (rip). 90

Chapter 1

Introduction

1.1 Motivation

While many modern defense techniques have been recently proposed, very few of them have been actually adopted mainstream or deployed in production. For this reason, attacks continue to gain ground on real-world applications whenever a new vulnerability is found. Attack vectors have increasingly become more versatile in their attack capability evolving from direct memory disclosure [15, 117], to indirect memory disclosure [14, 114], to creating attack chains on the fly during runtime [121, 134]. In order to address the versatility in various attack vectors, exploit mitigation techniques have thus far been correspondingly matching attacks by providing more coverage of application code. Adequate security is challenging to accomplish in a practical manner when so many sensitive and moving parts required to be protected. In order to provide this protection, defense techniques have gravitated towards more complex designs including re-randomization [21, 141], data integrity [65, 84, 122], memory safety [23, 30, 33, 99, 139, 145], and control-flow integrity [4, 44, 60, 77]. Nonetheless, some straightforward approaches like information hiding [10, 57, 104], debloating [83, 111, 146],

and system call filtering [32, 47, 103] have been found along the way. Regardless if the exploit mitigation strategy is complex or simple, many techniques are missing key properties of practicality. It is challenging in defense design to skillfully balance performance, strong security, scalability, and coverage, all at the same time. Strong defenses that have large coverage such as CFI [4] or DFI [19] often have proportionally high performance overhead and require significant hardware resources to operate. By contrast, defenses that has negligible or no performance overhead, such as Address Space Layout Randomization (ASLR) [126] and system call filtering (*e.g.*, seccomp) [72, 75] instead offer a weaker security level or are fragile to configuration changes or unexpected user input.

The following helps explain the motivation for practical exploit mitigation design by introducing code re-use attacks, modern defense approachs, and laying out the challenges in practical defense design.

1.1.1 Attacks, How they Attack

From the start of the intricate relationship between attack development and counter-active defense design, it has always been a race. 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 a high level view of a conventional code re-use attack procedure. First, most attacks assume there is an inherent vulnerability in a targt application. For the most part, these vulnerabilities originate from human error, design flaws, or unintentional bugs in the form of memory vulnerabilities or unvalidated user input or error checking which can then create an initial openning for a target victim application. This vulnerability allows for the memory disclosure of the entire code layout, allowing an attacker to locate the code components they need to chain together in order to

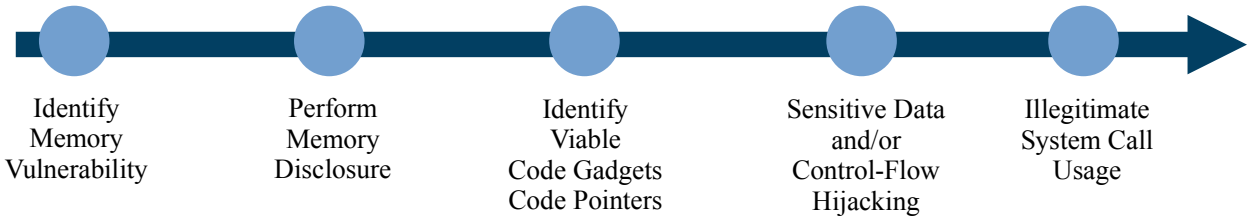


Figure 1.1: Conventional attack procedure to accomplish a code re-use attack

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 Major Defense Archetype	Debloating	Memory Safety	Information Hiding	Re-Randomization	Data Integrity, Control Flow Integrity	System Call Filtering
Defense Design Examples	Piece-wise [111] RAZOR [110] CHISEL [56] Nibbler [6] TRIMMER [7] PacJam [106]	HardBound [33] SoftBound [99] BOGO [145] ViK [23] Oscar [30] CHERI [139]	Oxymoron [10] Readactor [25] HideM [48] NEAR [140] HeisenByte [124] kRX [108]	**MARDU** [68, 70] Shuffler [141] CodeArmor [21] ReRanz [136] TASR [13] RuntimeASLR [92]	DFI [19] CPI [84] VIP [65] CFI [4] uCFI [60] OS-CFI [77]	**BASTION** Saffire [98] sysfilter [32] Temporal [47] ABHAYA [103] Shredder [97]

Table 1.1: Table showing the corresponding Defense Design Archetypes for each attack step.

form a viable attack. As shown by Snow *et al.* [121], a single memory disclosure, or even an indirect memory disclosure (*e.g.*, Blind ROP [14]) can allow an attacker to acertain all necessary information. Once the attacker knows the specific layout, they can manipulate each code component at will, via corruption of code pointers and variable values. Last but not least, a system call is leveraged to finalize an attack chain and achieve their intentions. Malicious attacker intentions can include vectors 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 Defenses, How they Defend

Instead of being spread thin attempting to protect everything, modern defense techniques have specialized into a few distinct archetypes. Specifically, these archetypes originate from understanding the steps necessary to complete a code re-use attack, and then attempting to

stop the attack at that specific step and preventing it from proceding any further. Recall Figure 1.1 shows a high level view of a conventional code re-use attack procedure. We reorganize this information in order to better show the relationships between attack angles and the corresponding defense used to block a particular step. Table 1.1 denotes the archetype that addresses code re-use attacks at each particular step as well as notable defenses that implement the archetype design. Each major defense archetype has its own respective advantages and disadvantages. It is important to note that some defense archetypes are complete and deterministic, however they often expensive performance-wise. Others have negligible performance impact but have trouble defending against more complex attacks. In reality, very few proposed defense techniques are able to excel in all aspects of practicality: performance, security, scalability, and reliability.

1.1.3 Challenges in Practical Defense Design

Regardless of the major archetype a defense employs, each experiences different challenges in practicality to various degrees when it is implemented. Here, an explanation is provided of the common challenges faced when attempting to create a new defense design that is practical.

- **Performance Tradeoffs** Tracing and updating 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 runtime, either for validation purposes (*e.g.*, data integrity) [19, 84] or to maintain program semantics due to the constant churn of code (*e.g.*, randomization) [13, 92, 141]. Ironically, in the case of randomization [42, 69, 136], the defense technique must keep track of code chunks in the same way the attacker needs to track the same code chunks in an attempt to

overcome the defense.

These factors often cause a defense technique to incur slowdown upon the target application being protected.

- **Security Tradeoffs** Security tradeoffs specifically refer to defense techniques that intentionally limit themselves in the provided strength or coverage of their security design in order to achieve a different practical property, notably preformance or scalability. Defenses that implement this strategy do raise the bar for the level of security provided somewhat. However, by limiting or loosening the security constraints imposed, this can potentially break the defense once a flaw is identified. In this case, the defense can protentially be side-stepped in practice, making the defense essentially be null at best, and give a false sense of security at worst. Prime examples of this include defense techniques such as Address Space Layout Randomization (ASLR) [126] and seccomp [72, 75]. ASLR performs a single load-time randomization of the starting base address for the runtime placement of the code region. It was soon identified that a single memory disclosure vulnerability could nullify the randomization imposed on the application [15, 116]. Seccomp instead tries to reduce the exposed attack surface for a given executable by disabling all unused system calls. However, it was found that the specific system calls that are often targeted by attackers (*e.g.*, execve, mprotect, mmap) cannot be disabled as they are legitimately required for process creation.

  Security tradeoffs are especially dangerous to users as defense designs could give users a false sense of security. When designing a new defense, security strength should always be the forefront 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 the defense's runtime. For example, some defenses [10, 53, 124] employ back-

ground monitors or usage of 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 schemes. This also includes limited ability to perform memory de-duplication to share shared code/data efficiently. For instance, very few fine-grained randomization approaches are capable of code sharing besides MARDU[69] and Oxymoron [10]. Other defenses increase memory use by having multiple permutations of code layout loaded into runtime memory [31] or additional hardened versions of functions [98]. Sharing of debloated code [6, 111] could break applications due to intentional inconsistencies performed by the defense mechanism. A defense design with limited scalability will likely result in being a very resource-draining defense taking away host system resources that could otherwise be used for useful work. Remedies for this challenge are indeed very difficult to incorporate as early design decisions will likely 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 defense design. Ideally, when a defense is deployed on a target application, original application semantics should be maintained throughout runtime, and a defense should never break the application. Many, if not all defense designs rely on some form of analysis of a 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. In static analysis approaches, as in debloating [6, 7, 83, 111, 113], can inadvertly miss valid control-flow paths or remove functions needed in other valid application configurations. Deriving precisely all necessary information from static analysis is undecidable and remains an open research area [85]. In dynamic analysis, approaches rely on various metrics such as machine learning [56],

provided test cases [45, 146], or runtime inputs or control-flow [35, 101, 131], which may not be sufficient to exercise all applicable code paths or functions. Additionally, defenses 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, defenses have had to make a difficult decision regarding how much fragility their defense design inherently possesses. Some techniques overapproximate their analysis in order to circumvent fragility. As a result, this often leads to an defense that still gives an attacker just enough wiggle room to still evade the defense. For example, in Control Flow Integrity techniques [4, 35, 100, 101, 131, 144], overapproximated analysis can bring on code pointers who have a set of valid target functions, commonly called an equivalence class [17], instead of enforcing the one true legal target [60]. Those defenses who do not overapproximate must risk breaking during runtime or illegitimately terminating because of a false-positive attack attempt. Applying a different input configuration can render a protected application inoperatable unless it is re-analyzed and re-compiled with the corresponding defense policy for a new configuration. Similarly, techniques such as debloating [6, 56, 83, 110, 111, 118], 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. Static and dynamic analysis will not improve unless the defense community is able to extract more context clues to gain insight about expected and legitimate program semantics. Fragility can be potentially resolved with an effort to deduce new contexts still hidden within code in order to derive new potentially helpful information in order to reduce fragility. Further research into various yet unexplored contexts in every aspect regarding code

component may help eliminate fragility as a challenge in defense design.

1.1.4 Goals

The described fundamental challenges in practical defense 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 defense approaches.

This thesis proposal seeks to address the following questions:

- Is it possible to resolve the performance, scalability, fragility, and security challenges in order to achieve practical defense design? If so, how?

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

This thesis proposal presents two designs and one full implementation and evaluation to show that practicality is possible and should be considered more seriously in defense design. Specifically, these two technical defense designs use the insight of deriving the core aspects absolutely essential to fulfull each defense's protection scope.

For MARDU, this is in the form of on-demand randomization rather than arbitrary active randomization, as well as using trampolines to effectively partition code to be capable of being shared. For BASTION, this is the form of stritly enforcing its defense from the perspective of system calls, such that all verification checks only occur at system call invocations

and only data associated with system calls is traced.

1.2 Contributions

The overarching goal of this thesis proposal is to design strong defense mechanisms that are practical enough to be deployed mainstream and capable of denying advanced attacks. This thesis proposal specifically defines the following four axioms for considering a defense mechanism to be practical.

- **Low Performance Impact** This thesis aims to design defense mechanisms such that their runtime overhead is acceptable. 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 try to minimize and aptly balance performance tradeoffs during its design. Specifically, this thesis strives to design practical defense techniques that have a realistic performance overhead of no more than 10%, and ideally less than 5%.
- **Scalable** In order to be considered scalable, this thesis aims for its defenses to be cognizant of its resource usage. Specifically, this thesis proposal intends for a practical defense to incur no or minimal usage of additional system resources. This includes usage of memory and CPU cores that could otherwise be utilized for useful work by applications on the host machine.
- **Reliable** Another property that users worry about when considering a defense mechanism is reliability. This specifically refers to a defenses 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 thesis proposal intends for practical defense design to not have such flaws.

- **Strong Security Guarantees** Last, but certainly not least, a practical defense design should provide strong security guarantees so that when it is deployed, the defense mechanism performs as expected. This axiom is the most important to consider when designing a defense, however the challenge that is often presented is that the potency of security is often proportional to the performance incurred and resources required. Not many techniques think outside the box in order to work around this crucial dilemma.

This thesis proposal presents two research thrusts to showcase the viability of designing practical defense mechanisms while following the above defined axioms. The first thrust 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 a costly mechanism, where most techniques even require stopping-the-world in order to perform re-randomization and update all necessary data to maintain semantics during runtime. The second thrust 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 [75], 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 thesis proposal makes the following contributions:

- The design and implementation of MARDU, a on-demand, scalable, re-randomization

defense is presented. The MARDU defense framework is made up of a custom LLVM compiler as well as a modified kernel. MARDU realizes a competitive performance tradeoff by only re-randomizing code at program start and when suspicious attacker behavior is observed during runtime. Additionally, MARDU is capable of sharing code; because of this its re-randomization scheme is scalable without requiring additional memory per-process allowing randomization to be applied system-wide.

- The design of BASTION, a specialized defense for legitimiate system call usage is presented. BASTION is founded on the insight that system calls can be considered a critical lynchpin in many attack models. Although past, current, and future attacks may vary in complexity and approach, it has been observed that can be boiled down to requiring system calls to complete their attack, whether it be control-flow hijacking or privilege escalation. By realizing that attacks must interact with the operating system, just as legitimate application operations do, BASTION is able to confidently omit expensive tracing and constant program interference for detecting malicious operations. BASTION instead narrowly focuses on fully encompassing system calls with strong invariants to impose legitimate system call usage.

1.3 Summary of Post-Preliminary Exam Work

After the preliminary examination, this thesis proposes the primary future work to be the completion of this thesis's second research thrust, BASTION. BASTION will be a comprehensive defense framework for the enforcement of legitimate system call usage. In particular, BASTION will implement the enforcement of three newly proposed system call contexts. These contexts include: verifying that the correct call-type is used, verifying that a system call is reached via control-flow in an expected and legitimate way, and verifying that all

argument variables as well as all data-dependent variables for those arguments are genuine and unaltered. Currently a bare-bones prototype has been made for BASTION and allowed us to obtain a small preliminary performance evaluation.

This thesis also proposes a comprehensive security evaluation of BASTION. A thorough security evaluation is necessary in order to validate the claims of BASTION's security guarantees. This can allow BASTION to be appropriately compared against leading modern defenses strength for its unique defense approach. Additionally, this security evaluation will further substantiate that the specific code components that are ommited in protection by BASTION can actually be left as-is without protection without weakening the security of an application.

As mentioned above, a simple prototype has been created for BASTION. However, it currently only supports NGINX. Additional code implementation is necessary to handle analysis of more complex code scenarios. This thesis proposes studying an expanded set of benchmarks and real-world applications in order to more fully understand the performance profile for each of BASTION's proposed system call contexts as well as BASTION's full performance impact footprint. This could also allow the identification of beneficial program system call usage profiles.

1.4 Thesis Organization

This thesis proposal is organized as follows. Chapter 2 presents background information regarding this thesis proposal. This includes explaining relevant attack models, code components that are traditionally targeted by attackers, and modern defense technique archtypes that address these attacks. Chapter 3 describes MARDU, a re-randomization-based defense technique that presents solutions to challenges inherent in re-randomization defenses, such

as unattractive performance tradeoffs and the inability to share dynamically randomized code. To the best of our knowledge, MARDU is the first defense framework capable of re-randomizing shared code 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 work. Thereafter, Chapter 4 presents a proposal to secure legitimate system call usage as a means to further explore the possibility of building realistic deployable mitigations with strong security guarantees. This work specifically seeks the address the current lack of specialization regarding security-critical functions such as system calls. Chapter 5 outlines post-preliminary exam work that should be carried out. This also describes possible directions of future work that could be explored. Chapter 6 concludes, summarizing this proposal's overarching themes and reiterates its research vision.

Chapter 2

Background

This chapter goes over background information that is relevant for understanding and potential clarification of this thesis proposal. A major part of this thesis proposal involves understanding the relationship between attack models and modern attack defenses proposed by the security community. Additionally, understanding how each attack model and major defense technique archtype functions can also help understand this thesis proposals contributions. 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 can confidently address. Section 2.3 describes the major defense archtypes that are commonly 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.

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, are integral elements in both the C and C++ programming languages. 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 the 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 their inherent flexibility of pointers especially void pointers, they are a prime target to be corrupted and replaced with an attacker desired variable or function. 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 As identified by Chen *et al.* [20], non-control-data such as configuration data, user input, user identity data, and decision-making data are considered critical to software security. These are ordinary data variables in a program that are used to guide the intended flow of program execution. For example, this can 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, this data and variables associated are much more discrete. As mentioned by Chen *et al.*, these variables are not primary targets for attackers as they traditionally require much more work and hoops to be jumped through in order to enable them malicious use, but it is possible. Instead of targeting function pointers, these variables can also 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 snippets of assembly code that end with a return instruction. These snippets can perform a vast array of operations such as addition, subtraction, setting of variables, etc. Given enough gadgets present in an application, manipulation with gadgets can even be Turing complete [116]. 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 In C++, objects are an instantiation of a C++ class. When a class is defined, no memory is allocated. Instead, objects are used to perform object-oriented programming, such that data and functions that operate on an objects data are bound together. Then objects interact with one another by sending messages to each other and performing operations on the data within an object. C++ objects however can be counterfeited by an attacker. This code component is specific to the Counterfeit Object Oriented Programming (COOP) attack [115], where a attacker created counterfeit object with attacker populated virtual function pointer is introduced and then executed.

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 thesis proposal seeks to address with its research thrusts. We present attacks that are capable of being addressed with randomization-based defenses as well as attacks that commonly leverage system calls to tie in the second research thrust in this proposal. 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 made modern attacks 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 attack models, instead only those attacks relevant to this thesis proposal are presented. Specifically, all attacks variants listed below for the most part fall under the umbrella of code re-use. Additionally, malware is considered

out of scope for this proposal.

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 code gadgets within existing trusted software and manipulate the control flow as desired. ROP attacks rely on identifying the location 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. Once all needed code gadgets are located, the attack can be completed.

Just-In-Time ROP (JIT-ROP) Just-In-Time Code Reuse [121] works incrementally from a single code pointer vulnerability. Given a single disclosed code pointer obtained from a present memory vulnerability, JIT-ROP is then able to further leap frog and map as pages of memory as possible. Once more code is diclosed JIT-ROP is able to dynamically determine viable code gadgets and required system calls 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 [107, 120] and Blind ROP [14] 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 active code gadgets and abuse control flow vulnerability to execute the attacker payload.

Code Pointer Offsetting If code pointers are not protected from having arithmetic operations applied by attackers code can be susceptible to a code pointer offsetting attack. Partially overwriting of the low-order byte of code pointers can be leveraged such that they

point to a relative offset within the randomized region [137]. This attack takes advantage of unprotected code pointers so that arithmetic operations can be performed over pointers, allowing access to other ROP gadgets.

Privilege Escalation Attacks Privilege escalation attacks attempt to illegitimately gain elevated rights and privileges beyond what is intended for a given user on a host machine. If achieved, the attacker will then have illegitimate administrator root access to the machine, allowing them to perform malicious actions at will. While attack class of privilege escalation is very broad in terms of methodology to be achieved, this thesis proposal specifically focuses on addressing vertical privilege escalation originating from available vulnerabilities and exploits found in a victim application.

Control Data Attacks Attacks that attempt to compromise a victim program's control-flow to be redirected to an attackers exploit payload. Commonly, this is done through the corruption of code pointers which dynamically point to functions within an a program.

Non-Control Data Attacks Manipulating a program's data, rather than its control flow, can be enough for a viable exploitation of a victim program. Non-Control Data attacks can be used to mount privilege escalation or leak sensitive information as demonstrated by Chen *et al.* [20] and Hu *et al.* [59]. Instead of attempting to leverage code components that are typically protected by standard defenses, this attack leverages non-control data (Section 2.1). Note that non-control data attacks are considered out of scope for this thesis proposal and will not be discussed further.

2.3 Modern Defense Techniques

This section presents a rough categorization of present day defense 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.

Control-Flow Integrity Control-Flow Integrity, also referred to as CFI, 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. Ever since the first publication of CFI [4], many subsequent iterations and variations regarding protection of a target programs code pointers. Most recently, uCFI [60] have introduced techniques to help reduce the size of equivalence classes to apply stronger constraints to each code pointer.

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 [145] provides memory safety for all dynamically allocated data, CPI [84] guarantees the integrity of all code pointers, VIP [65] covers virtual function tables and heap metadata in addition to code pointers, and DFI [19] 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 [41].

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, code region, function, basic-block, or assembly instruction granularity. Re-randomization is an active approach that is performed throughout runtime upon a target program. Re-Randomization can be triggered by a variety of different metrics including set-time intervals [21, 29, 141], system call history [13, 92, 136], or suspicious runtime behaviors like a crashed worker thread [68, 70]. Additionally, re-randomization can be specialized to be conducted on different pieces of information varying from the code layout [21, 136, 141], code pointer values [13], or the entire memory address space itself [49, 92]. 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 thesis proposal discusses one such re-randomization technique in Chapter 3.

Debloating Debloating is the process of purposely removing code, specifically, code determined to be unused, in order to reduce the viable attack surface of an application. Debloating works by removing as many possible code gadget snippets from the application code as possible. By *physically* removing as many possible code gadgets as possible, this defense potentially makes creation of certain attack vectors impossible since the necessary code gadgets to create an attack are missing after debloating is performed on the target

executable. Debloating can configuration-driven, as in Koo *et al.*. [83], via static analysis of code dependencies as in Piece-wise [111] and Nibbler [6], or with user test cases [45, 110]. Notably, because of this techniques 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.

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. This defense reduces the total kernel surface exposed to the application who may be the target of an attacker to become a bad actor within the host leading the compromise of the host machine. 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 [72, 75] is currently deployed on Linux as an available system call filtering technique for users. Most techniques proposed thus far in literature, such as Confine [46], sysfilter [32], and ABHAYA [103], only seek to automate and help in the derivation of viable system call filtering policies for users. A subset of system call filtering is system call specialization, which adds further constraints regarding system call usage, such as a set of legal static argument values for a given system call. Temporal system call specialization [47] enforces temporal policies that only allow certain system calls during certain phases of execution. Additionally, this thesis proposal introduces BASTION, a design for a new approach to achieve more complete system call filtering in Chapter 4.

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) [126] and similar finer-grained variants [57, 78, 104]. 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 [48, 108, 124, 140] and/or code pointer tables (*e.g.*, the Global Offset Table (GOT)) [28], 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 thesis proposal and will not be discussed further.

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 [30, 86, 130] provide memory safety via secure heap allocation [36, 119], and enforce bounds-checking that leverage spatial or temporal constraints [33, 99, 145]. 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) [126]. Examples like *return-oriented programing (ROP)* and ret-into-libc [116], 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 [121]. Execute-only memory (XoM) was introduced to prevent direct memory disclosures, by enabling memory regions to be marked with execute-only permissions [25, 28]. However, code inference attacks, a code reuse attack variant that works by indirectly observing and

deducing information about the code layout circumvented these limitations [14]. 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 [13, 92].

This fostered the next generation of defenses, such as Shuffler [141] and CodeArmor [21], 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 [10]. 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 [128], and iOS as well as having support in compilers like LLVM [125] and gcc [127]. 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 [21, 141] or the amount of leaked data [136], 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) [62]. 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 [25, 26] 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 [141] and ReRanz [136], 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 [13, 21, 42, 136, 141]). 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.

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 [136, 141] 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 [24, 57, 58, 78, 81, 104, 138]	Fine	×	×	N/A	×	✓	0.4%	6.4%	×	✓
	Oxymoron [10]	Coarse	×	×	N/A	×	✓	2.7%	11%	✓	✓
	Pagerando [27]	Coarse	×	×	N/A	×	✓	1.09%	6.5%	✓	✓
	Isomeron [31]	Fine	●	●	N/A	▲	×	19%	42%	×	×
Load-time+XoM	Readactor/Readactor++ [25, 28]	Fine	●	×	N/A	▲	✓	8.4%	25%	×	✓
	LR 2 [16]	Fine	●	×	N/A	▲	✓	6.6%	18%	×	✓
	kR^X [108]	Fine	●	×	N/A	▲	✓	2.32%	12.1%	×	✓
Re-randomization	RuntimeASLR [92]	Coarse	×	●	N/A	×	×	N/T	N/T	✓	✓
	TASR [13]	Coarse	▲	●	×	×	×	2.1%†	10.1%†	×	×
	ReRanz [136]	Fine	▲	●	×	▲	✓	5.3%	14.4%	×	×
	Shuffler [141]	Fine	●	●	×	▲	×	14.9%	40%	×	×
	CodeArmor [21]	Coarse	●	●	×	×	×	3.2%	55%	×	×
Our Approach	MARDU	Fine	●	●	●	▲	✓	5.5%	18.3%	✓	✓

3.2.1 Attacks against Load-time Randomization

Load-time Randomization without XoM. Code layout randomization techniques such as coarse-grained ASLR [126] and fine-grained ASLR [10, 24, 31, 57, 58, 78, 81, 104, 138] 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 [121] 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 [124, 140] or execute-only memory (XoM) [11, 16, 21, 25, 28, 48, 108, 124, 140]. 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 [14] or other inference attacks [107, 120]. BROP infers code contents via observing differences in execution behaviors such as timing or program output while other attacks [107, 120] defeat destructive code read defenses [124, 140] 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.

3.2.2 Defeating A1/A2 via Continuous Re-randomization

Continuous re-randomization techniques [13, 21, 42, 49, 92, 136, 141] 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 [21, 141] shuffle the layout periodically by setting a timing window. For example, Shuffler [141] triggers re-randomization every 50 msec (< network latency) to counter remote attackers, and CodeArmor [21] can set re-randomization period as low as 55 μsec.
- **System-call history:** Techniques [13, 92, 136] shuffle the layout based on the history of the program's previous system call invocations, *e.g.*, invoking fork() (vulnerable to BROP) [92] or when write() (leak) is followed by read() (exploit) [13, 136].

By code pointer semantics:

- **Code address as code pointer:** Techniques (ASR3 [49] and TASR [13]) 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 [141]) or the address of a function trampoline (ReRanz [136]), 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 [21]. 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 [21, 42, 49, 141], amount of transmitted data by output system calls [136], or I/O system call boundary [13] 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 [13, 21]. An attacker may trigger a vulnerability to apply arithmetic operations to an existing code pointer. Particularly, in techniques that directly use a code address [13] or a code offset [21], the target could be even a ROP gadget if the attacker knows the gadgets offset beforehand. Ward *et al.* [137] 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 [13, 21, 141]. 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) [62], 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 [79], Meltdown [90]) 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 [10] and PageRando [27] 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 [21, 136, 141] 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 [21, 25, 28, 141] 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 [10] 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 attempt 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 [107, 120]. We use Intel MPK [62] 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

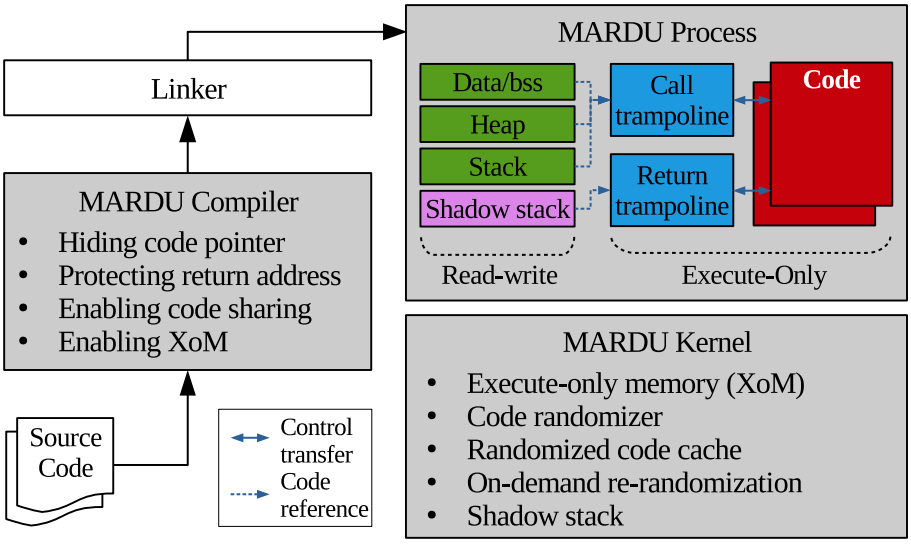


Figure 3.1: Overview of MARDU

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 [13, 21, 92], 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 the return trampoline address ❹. Finally, the return trampoline

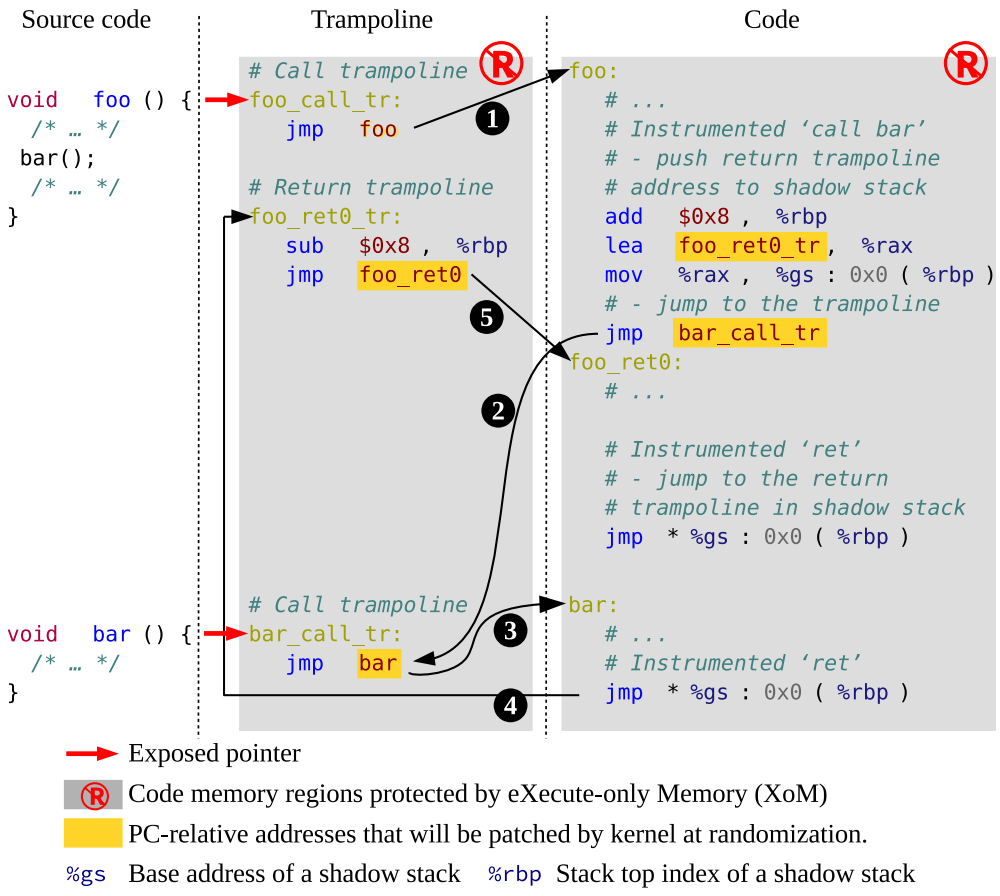


Figure 3.2: Illustrative example executing a MARDU-compiled function foo(), which calls a function bar() and then returns.

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) [82], 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 [40].

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 re-claims 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

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

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 [18]. 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

[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.

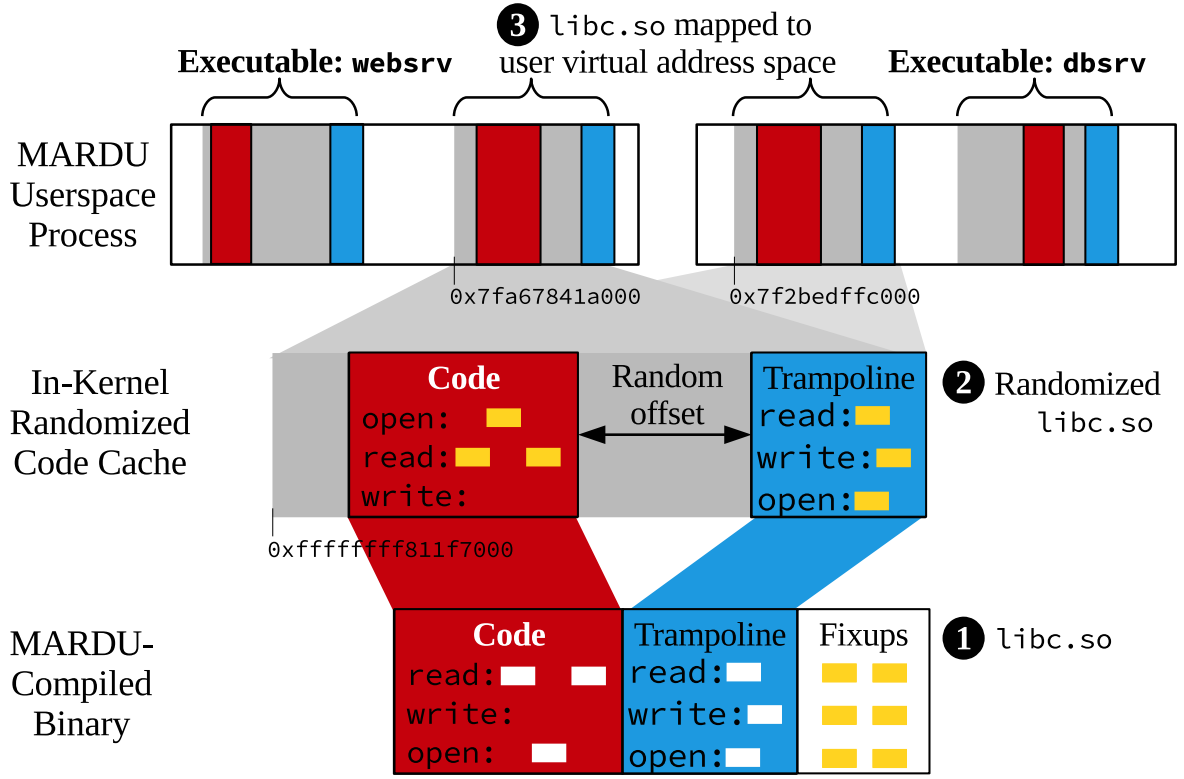


Figure 3.3: The memory layout of two MARDU processes: websrv (top left) and dbsrv (top right). The randomized code in kernel (0xffffffff811f7000) is shared by multiple processes, which is mapped to its own virtual base address (0x7fa67841a000 for websrv and 0x7f2bedffc000 for dbsrv).

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 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).

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

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 [62][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 [124] that require system resources to enable virtualization as well as have inherent overhead from nested address translation,

[4]As of this writing, Intel Xeon Scalable Processors [63] and Amazon EC2 C5 instance [8] support MPK. Other than x86, ARM AArch64 architecture also supports execute-only memory [9].

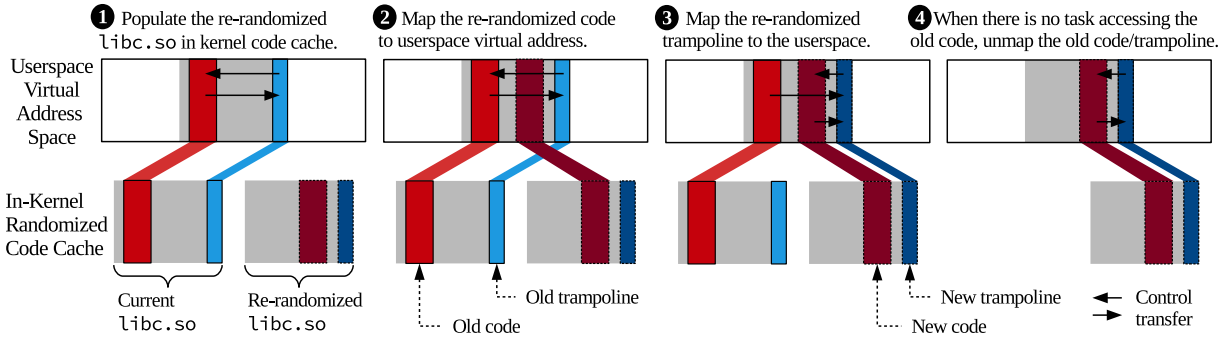


Figure 3.4: Re-randomization procedure in MARDU. Once a new re-randomized code is populated ❶, the MARDU kernel maps new code and trampoline in order ❷, ❸. This makes threads crossing the new trampoline migrate to the newly re-randomized code. After it is guaranteed that all threads are migrated to the 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.

our MPK-based design does not impose such runtime overhead.

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

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

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 [1], 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.

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 [18].

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. This benchmark suite has realistic compute-intensive applications, ideal to see worst-case performance overhead of MARDU. We tested all 12 C language benchmarks using input size *ref*; we excluded C++ benchmarks as our current prototype does not support C++. We choose SPEC CPU2006 over SPEC CPU2017 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. All programs are compiled with optimization -O2 and run on a 24-core (48-hardware threads) machine equipped with two Intel Xeon Silver 4116 CPUs (2.10 GHz) and 128 GB DRAM.

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 [134] 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, includ-

ing BROP [14], clone-probing [92], and destructive code read attacks [107, 120] 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) [114], 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 [38, 51, 102]. 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 [38, 43, 51, 80, 102]). 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 [134]. 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 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 [21, 136, 141]. Although MARDU allows attackers to reuse function point-

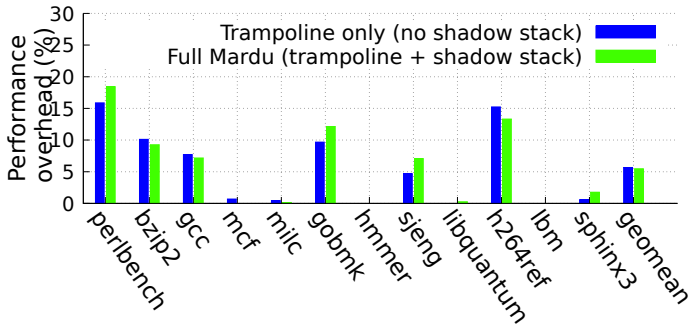


Figure 3.5: MARDU performance overhead breakdown for SPEC

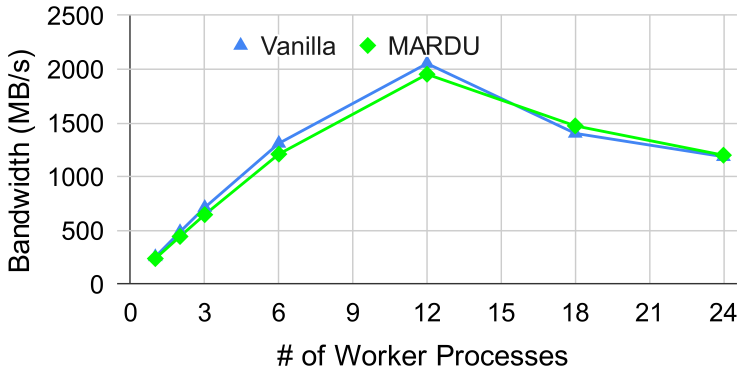


Figure 3.6: Performance comparison of NGINX web server

ers 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 [141] and CodeArmor [21], 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 [141] and CodeArmor [21], 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 [13] shows a very practical average overhead of 2.1%; however, it has been reported by Shuffler [141] and ReRanz [136] 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 [50] 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 [141] 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.

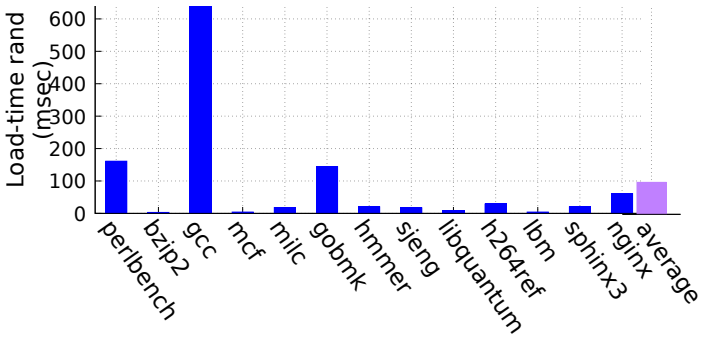


Figure 3.7: Cold load-time randomization overhead

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

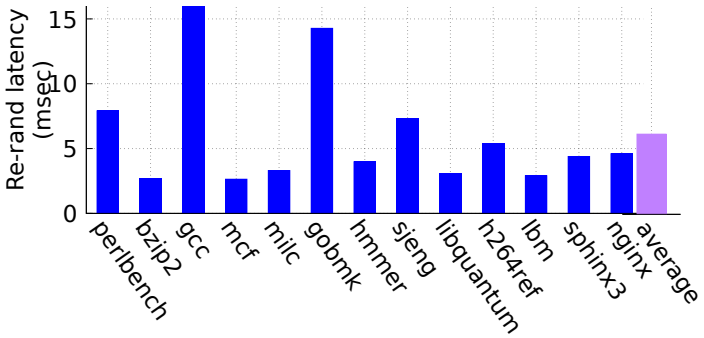
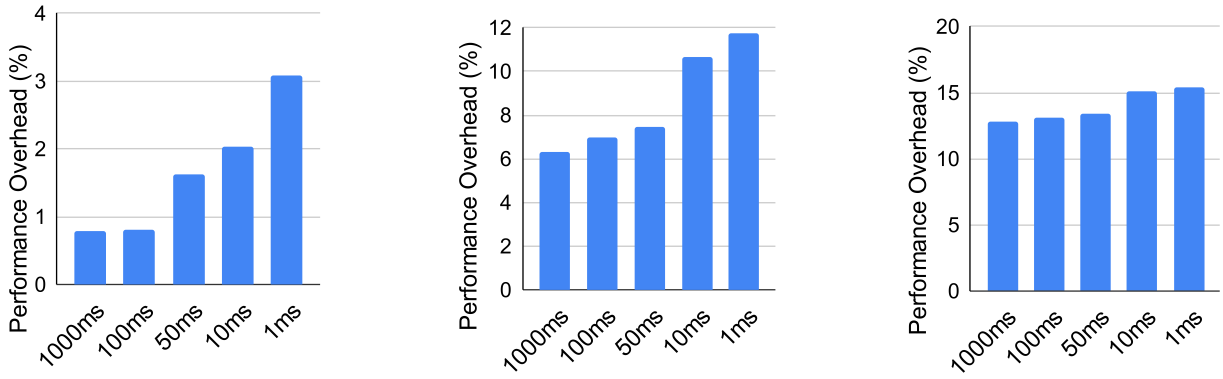


Figure 3.8: Runtime re-randomization latency

layout, and reclaiming 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 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



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

Figure 3.9: Overhead varying re-randomization frequency

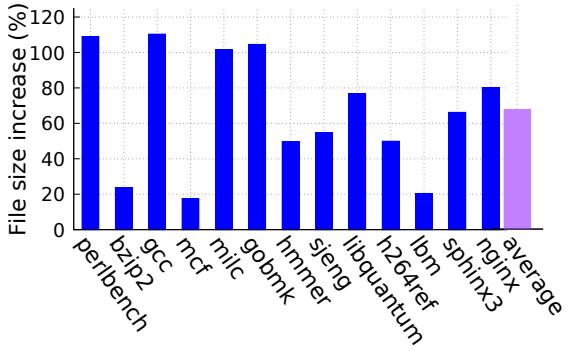


Figure 3.10: File size increase with MARDU compilation

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

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

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 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.* [10] and Ward *et al.* [137] 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:

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

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 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.

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.

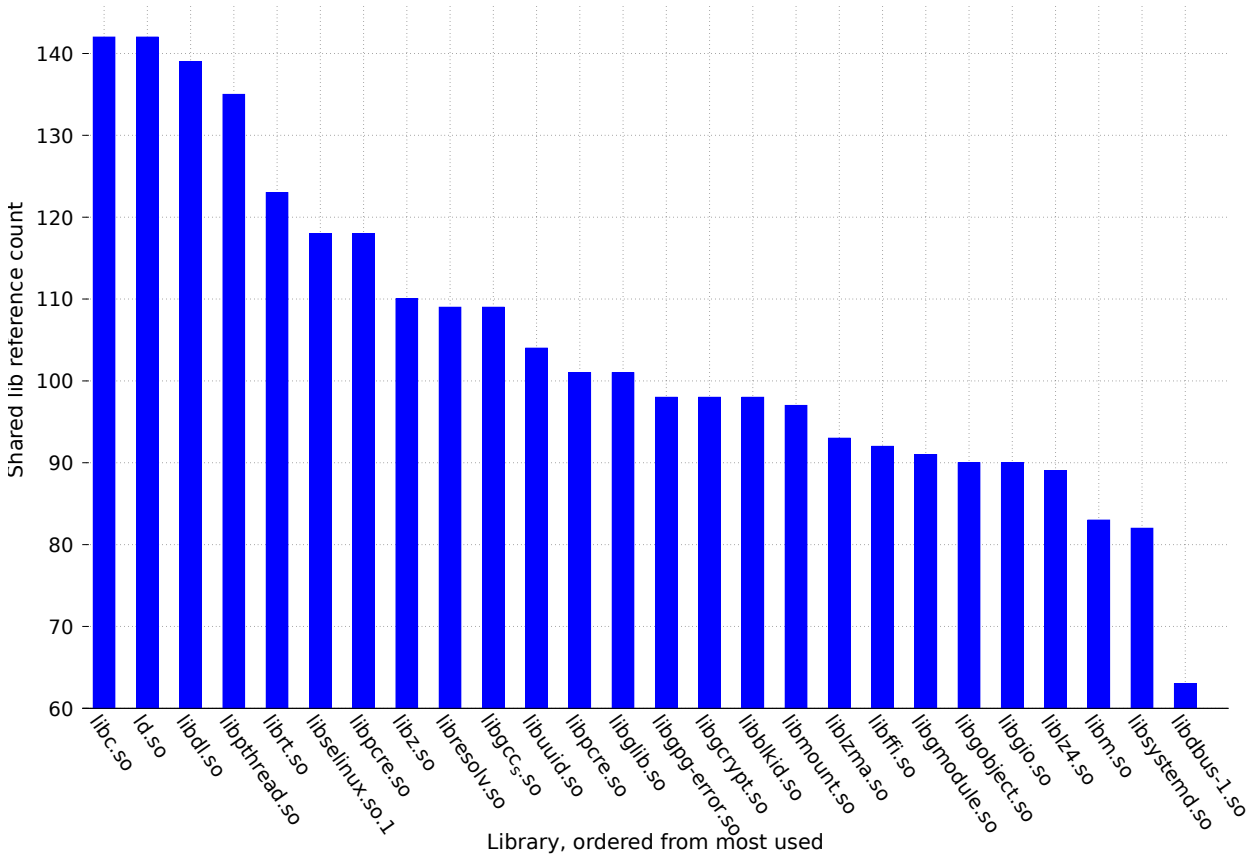


Figure 3.11: Top 25 shared libraries with their reference count on our idle Linux server ordered from most linked to least linked libraries.

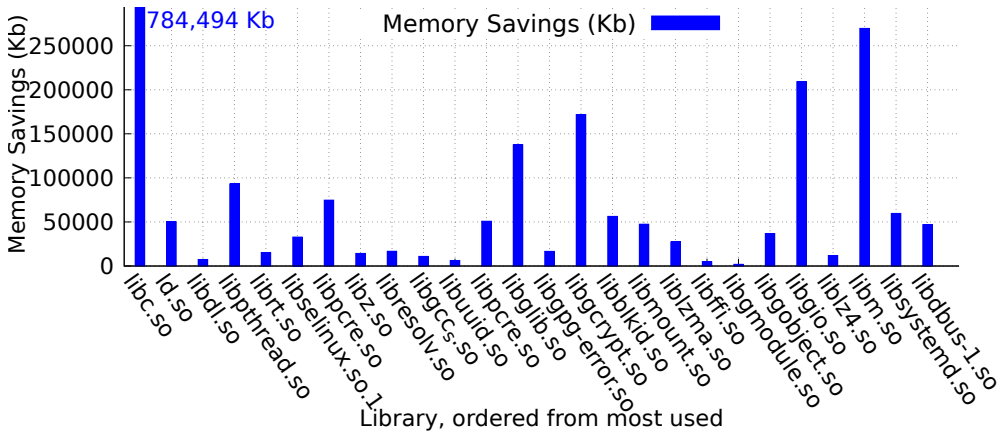


Figure 3.12: Estimated runtime memory savings with shared memory MARDU approach for the top 25 most linked libraries on our idle Linux server.

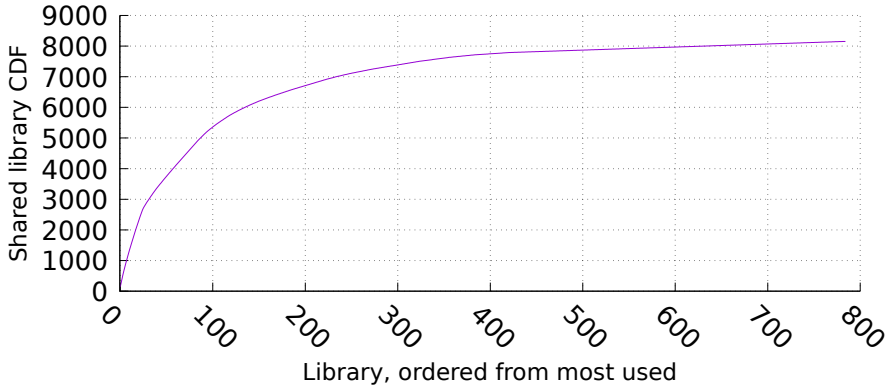


Figure 3.13: CDF of shared library occurance on a idle Linux server.

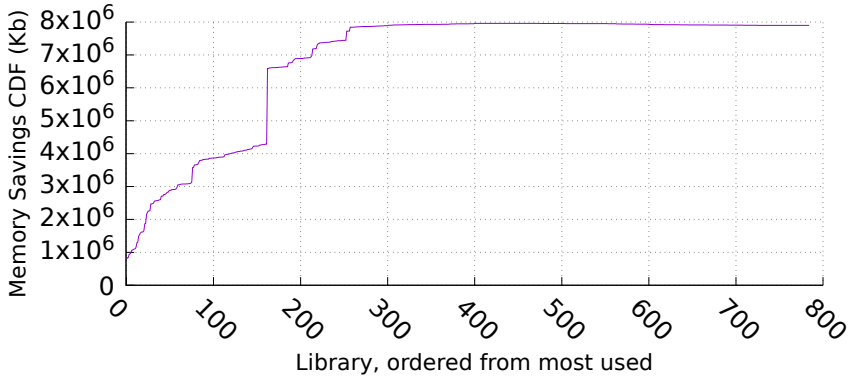


Figure 3.14: CDF plot of estimated runtime memory savings with MARDU's shared memory approach.

A potential way to enable this is to apply techniques that can retrieve precise disassembly from a given binary, such as BYTEWEIGHT [12], to identify possible call targets. A more recent innovation, Egalito [142], 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 [34], 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 [54] or ERIM [129] 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 [21, 141], *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 [141].

Another possible solution to prevent these attacks could be pairing MARDU together with

control-flow-integrity (CFI) [4, 22, 44, 52, 53, 91, 100, 101, 105, 109, 127, 131, 133, 143, 144], code-pointer integrity/separation (CPI/CPS) [84], or other hardware-assisted solutions such as Intel's Control-flow Enforcement Technology (CET) [61] and ARM's Pointer Authentication Code (PAC) [112]. 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 [59] 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 of the computing landscape. They allow reading and writing from files, managing processes, and maintaining network connections. System calls are the chief mechanism that provide an interface between a process and the host operating system (OS) to request services. At the same time, system calls are a core component of various attack vectors (*e.g.*, code reuse, privilege escalation) enabling attackers to utilize OS services to complete their attack.

Thus, system calls in particular have received renewed interest in the security research community, with techniques such as debloating [6, 56, 110, 111, 118], system call filtering [32, 46, 47, 75], and system call sandboxing [67, 87, 135]. These techniques aim to minimize the available attack surface by disabling unused system calls. Some techniques explore

enforcing legitimate system call usage for those system calls that remain post-debloating. At the same time, fine-grained technqiues such as Control Flow Integrity (CFI) and Data Flow Integrity (DFI) remain largely unused due to inherent performance trade-offs. Crucially, all aforementioned defenses 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 three system call contexts necessary to provide a complete, sound, and sufficient defense to properly enforce the correct usage of system calls during runtime. These contexts collectively aim to form a stronghold for legitimate system call usage. Our first context allows system calls to be invoked with only the calling convention they are used with in the application. The second context only allows system calls to be invoked through a legitimate runtime control-flow path. The third and final context ensures argument integrity specifically for system call arguments.

The key idea of this work is to surround system calls with differentiated contexts to negate modern attacks leveraging system calls. This work is not attempting to apply an application-wide defense, but rather harden legitimate system call usage. Our first context is able to apply finer-grained system call filtering. Our second context is a scope-reduced CFI, purposely very narrow and concentrated to focus on system call related control-flow paths. Similarly, our third context provides data integrity, but only for system call arguments.

In order to fulfill all three system call contexts, it is neccesary to address the following challenges. For each of the three contexts, 1) what is the most meaningful relevant data to each system call, and 2) how to efficiently enforce these contexts.

Thus, we propose BASTION, a prototype defense system to address these challenges and enforce our three system call contexts. BASTION performs static offline analysis of all system calls within an application to derive metadata that is used in enforcement. We

develop a novel technique for our context analysis and enforcement. Specifically, all three contexts are obtained from the perspective of each individual system call, compared to callsites (as in CFI) or specific data (as in DFI). In BASTION, all metadata is associated with its particular system call. This includes considering programming paradigms such as function call types, control-flow paths, and argument tracing from the exclusive perspective of system calls rather than generic and application-wide.

We implement BASTION as a two part system, consisting of a custom compiler pass and a runtime enforcement monitor. Our BASTION runtime monitor uses metadata produced from the compiler analysis and minimal instrumentation to enforce all three system call contexts. Since system calls are often the most critical point in completing an attack, our prototype focuses on stopping attacks once they attempt to illegatimately invoke a needed system call. Using this defense paradigm, BASTION processes a minimal amount of metadata and intervenes at only the most critical point to effectively block an attempted exploit. Note that BASTION is not tailored to specific kinds of memory corruption vulnerabilities (*e.g.*, heap or stack-based) and is capable of addressing the generic attack class that leverages system calls.

We demonstrate the effectiveness and efficiency of BASTION applying it to several real-world applications.The minimal instrumentation and concentrated runtime interference by BASTION monitor produces negligible overhead of 0.17% on average for NGINX. Additionally, we breakdown the individual costs for each system call context and provide statistics regarding average call-depths and call frequency for system calls. This design is more light-weight compared to popular modern defenses. Therefore, our BASTION design is a practical solution to provide comprehensive hardening of system calls.

We make the following contributions in this paper:

- **Novel Specialized System Call Contexts** We propose three specialized, narrow system call contexts to collectively build a strong defense around legitimate system call usage in applications. Individually, each context aids to protect system call usage from a different perspective. Moreover, when all three system call contexts are applied collectively under a single defense, they are able to provide a complete and sound defense mechanism for legitimate system call usage. A system that enforces all three of our proposed contexts asserts that only permitted system calls are called in the correct manner, through a legitimate control-flow path, with authentic argument values.
- **Prototype of Specialized System Call Contexts** We design and implement a full prototype defense to enforce all three of our specialized system call contexts. Our prototype, BASTION, includes a custom compiler pass to analyze all system call usages present in a given application, perform any necessary instrumentation, and generate accompanying metadata. Our prototype also includes a runtime monitor which supports the complete enforcement of all static and dynamic aspects of each system call context. We develop a novel design for BASTION to be minimally instrusive in both instrumentation and runtime, to enable efficient enforcement of legitimate system call usage.
- **Performance Evaluation & Statistics** We evaluate our BASTION prototype using a varied set of real-world applications to evaluate its efficiency. Our evaluation and statistics survey show that a majority of system calls critical to OS operation are used sparingly and require minimal instrumentation for their protection. We also confirm that collectively using all three contexts in our BASTION prototype is sufficient to protect sensitive system calls from illegitimate use with minimal performance overhead.

The rest of this paper is organized as follows. We first explain our insights regarding system call usage and describe how current defenses address system call security (Section 4.2). We then detail each of our proposed specialized system call contexts (Section 4.3) and how they significantly improve constraints surrounding (sensitive) system calls. This is followed by our threat model and scope for our work (Section 4.4). We present our design for BASTION (Section 4.5, Section 4.6, Section 4.7) and provide implementation details (Section 3.5). We evaluate both security and performance of BASTION (Section 3.6), review related work relevant to BASTION (Section 4.9), and conclude (Section 4.10).

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

In the mindset of an attacker, they do not care about gadgets, ROP chains, or Turing completeness. Their sole purpose is almost always to reach the critical system calls that they need to complete their attack, compromise the application, and take over a machine with arbitrary execution [114, 134]. While there exists over 400+ system calls in recent Linux kernel versions [74], 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 play a vital role for attackers to be able to complete an attack. Recent works [22, 35, 60, 105, 131] make a (non-

exhaustive) list of sensitive system calls that should take precedence when building a new defense mechanism. These system calls include: mmap, mremap, remap_file_pages, mprotect, execve, execveat, sendmsg, sendmmsg, socket, clone, fork, ptrace, chmod, setgid, setreuid, setuid, bind, connect, accept4, accept, listen, and system.

With the insight that a majority of critical attacks need some form of an invocation of a sensitive system call, 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 have to 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 [6, 56, 110, 111, 118] reduce the attack surface by carving out unused code in a program binary. They leverage static program analysis or dynamic coverage analysis using user-provided 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 [72, 75] is a system call filtering framework. A system administrator can define an allowlist and 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 the actual argument constraining policy could be more permissive than necessary. Suppose that an application uses a given system call with two very different permissions flags (*e.g.*, read-only vs. read-executable). In this case, a system administrator should allow both permission flags so as not to break the application execution, thereby weakening the security guarantees.

Control-flow integrity (CFI). Enforcing the integrity of control flow could be one way to enforce the legitimate use of system calls in a program. CFI defenses [4, 17, 44, 52, 53, 60, 76, 77, 93, 100, 101, 127, 132, 144] aim to enforce the integrity of all forward and backward control flow of a program. CFI-style defenses perform program analysis to generate an allowed set of targets per-callsite, called an equivalence class (EC), defining legitimate control flow transfers. The size and accuracy of ECs are dependent on the analysis technique used to derive legal targets for a given callsite. Imprecise (static) analysis leads to a large EC, allowing attackers to bypass the CFI defenses [114, 134]. Enforcing an EC equal to one is ideal – *i.e.*, there is only one legitimate control transfer at a given moment in a program execution. However, CFI techniques maintaining an EC equal to one [60, 77] 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 [19] tracks data-flow of each and every memory access, instrumenting every load and store instruction. However, the effectiveness of DFI depends on the accuracy of the point-to analysis used to generate the data-flow graph. Also, DFI incurs significant performance overhead due to the invasive instrumentation [41].

4.3 Contexts for System Call Integrity

As discussed in the previous section, debloating and system call filtering approaches make a binary decision of whether a system call is allowed to be used in a program. Defining an allowlist, denylist, or argument values for a program is still too coarse grained and not effective to protect even commonly used sensitive system calls because the context of system call usage is not considered. CFI and DFI are general defense mechanisms, enforcing the integrity of either control or data flow. However, in general, they impose high runtime overhead and require significant system resources.

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 can prevent system calls from being weaponized. At a high level, enforcing these system call contexts implements a selective, narrower version of debloating, control-flow, and data integrity surrounding system calls. In this way, only program elements relevant to system calls are protected, 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 first blocks all unused system calls (*i.e.*, not-callable type). For the allowed system calls, it divides system calls into ones that can be called only from a direct call site (*i.e.*, directly-callable type) and ones that be called from an indirect call site via a function pointer (*i.e.*, indirectly-callable type). In our observation, it is very rare that a (especially sensitive) system call is called from an indirect call site so we can constrain accessibility of indirectly-callable system calls. Note that the call-type context and control-flow context (which will be introduced next) complement each other for efficient enforcement of control-flow integrity reaching a system call.

4.3.2 Control-Flow Context

Our system call *control-flow context* enforces that a (sensitive) system call can be reached and called only through a legitimate control-flow path 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 so that this context can be enforced 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. Our call-type context also compliments this control-flow 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 into (1) *direct argument* and (2) *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. On the other hand, the 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.

This context takes the expected argument type and its field into consideration for each individual argument for a given system call. For example, in Figure 4.1, 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 by this context.

Compared to traditional data integrity defenses like DFI [19], we limit the scope to system call arguments (and their data-dependent variables), and check the integrity of argument values [64] instead of enforcing the integrity of data-flow, thus reducing the runtime overhead to check this context.

4.3.4 Real-World Code Examples

We use two code snippets in NGINX web server [2] as running examples to demonstrate that various attack angles are possible and how all three proposed contexts can negate these attacks collectively. Note that the use of execve and mprotect in Figure 4.1 and Figure 4.2, respectively, is legitimate in NGINX, so debloating and system call filtering mechanisms cannot protect these system calls from being weaponized. The following attacks only assume the existence of a memory corruption vulnerability such as CVE 2013-2028 (NGINX) or CVE-2014-0226 (Apache).

(1) execve(). Figure 4.1 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 address) if a CFI-style defense is not deployed. Or she can corrupt the program name (ctx->path), its argument (ctx->argv), or its environment variables (ctx->envp) 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 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 a corruption of the global ctx object, and 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

```
1  // nginx/src/core/ngx_output_chain.c
2  ngx_int_t ngx_output_chain( ngx_output_chain_ctx_t *ctx, ngx_chain_t *in ){
3      ...
4      if ( in->next == NULL
5  #if (NGX_SENDFILE_LIMIT)
6          && !( in->buf->in_file && in->buf->file_last > NGX_SENDFILE_LIMIT )
7  #endif
8          && ngx_output_chain_as_is( ctx, in->buf ) )
9      { return ctx->output_filter( ctx->filter_ctx, in ); }
10     ...
11 }
12
13 // nginx/src/os/unix/ngx_process.c
14 static void ngx_execute_proc( ngx_cycle_t *cycle, void *data ){
15     ngx_exec_ctx_t  *ctx = data;
16     // Legitimate NGINX usage of execve system call
17     if ( execve( ctx->path, ctx->argv, ctx->envp ) == -1 ) {
18         ngx_log_error( NGX_LOG_ALERT, cycle->log, ngx_errno,
19             "execve() failed while executing %s \"%s\"",
20             ctx->name, ctx->path );
21     }
22     exit(1);
23 }
```

Figure 4.1: Legitimate use of the execve system call in NGINX.

control path to ngx_execute_proc() is rarely used in NGINX, only when an NGINX 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 the control-flow hijacking of the execve system call in NGINX. Likewise, our argument integrity context can also detect the corrupted system call arguments to block this attack proposed by Control Jujutsu [39].

(2) mprotect(). Figure 4.2 is another NGINX code snippet that can be used maliciously. Here, the statement v[index].get_handler(...) is found within the function ngx_http_get_-indexed_variable(). This statement is intended to be used a generic handler for NGINX indexed variables and 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

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

Figure 4.2: Snippet of NGINX code that can be compromised by an attacker to achieve Attack 2, to call the mprotect system call elsewhere in the code base using a vulnerable code pointer v[index].get_handler().

the v[] array to be reached.

An attacker can illegitimately reach and 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 bypass both code and data pointer integrity (*e.g.*, CPI [84]) as described in the Newton attack framework [134]. Instead of corrupting code and data pointers, the attack relies on finding a callsite that is capable of being manipulated by non-pointer values. Here specifically, 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 attacker controlled arguments.

Even here, our control-flow context in tandem with the call-type context can easily detect the control-flow hijacking reaching the mprotect system call, whereas CFI and CPI type defenses would not detect this attack. Also, our argument integrity context can detect that the leveraged variables r, &r->variables[index], and v[index].data are all illegitimate and never used by any legal system call invocation.

4.4 Threat Model and Assumptions

We focus on thwarting a class of attacks exploiting one or more (sensitive) system calls in their attack chain. 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) [71, 96] and (coarse-grained) Address Space Layout Randomization (ASLR) [126] – are deployed on the host system. Hence attackers cannot inject or modify code due to DEP. We assume that the hardware and the OS kernel are trusted, especially secomp-BPF [75] and the OS's process isolation. Attacks targeting OS kernel and hardware (*e.g.*, Spectre [79]) are out of scope.

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 [22, 35, 44, 60, 105, 132], we choose 22 sensitive system calls – mmap, mremap, remap_file_pages, mprotect, execve, execveat, sendmsg, sendmmsg, socket, clone, fork, ptrace, chmod, setgid, setreuid, setuid,

bind, connect, accept4, accept, listen, and system – which could be used for arbitrary code execution and information leakage. Note that this list is configurable and can be extended to include other system calls.

As illustrated in Figure 4.3, BASTION consists of two components: (1) a BASTION-compiler pass, which performs context analysis, instrumentation, produces a BASTION-enabled binary, and generates the program's context metadata, and (2) a BASTION monitor, which enforces the program's system call contexts during runtime. We now explain the BASTION compiler (Section 4.6) and the BASTION runtime monitor (Section 4.7) in detail.

4.6 BASTION Compiler

Our BASTION compiler derives the necessary information used for enforcement of our system call contexts. It also performs light-weight instrumentation of the program using BASTION's library APIs for dynamic tracking of specific data relevant to system call arguments. We now describe our program analysis and instrumentation for each context.

4.6.1 Analysis for Call-Type Context

To enforce the call-type context, 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 looks for all call instructions. If a system call is a target of a direct function call, BASTION compiler classifies it into a directly-callable tpe. If the address of a system call is taken and used in the left-hand-side of an assignment, this system call can be

Figure 4.3: The design overview of BASTION. At compile time, BASTION analyzes a program and generates the program's context metadata. For the call-type and control-flow contexts, BASTION generates static metadata information. For the argument integrity context, BASTION generates static metadata information to check static argument values (*e.g.*, constant) and instruments the program to track its dynamic argument values. At runtime, the BASTION monitor catches all invocations of sensitive system calls and verifies all three contexts of the system call by checking the generated metadata together with dynamic runtime information.

used as an indirect call target, and thus BASTION compiler classifies it into an indirectly-callable type. Note that a system call can be both directly-callable and indirectly-callable. All other system calls not belonging to either types are not-callable.

After the analysis is completed, BASTION compiler generates metadata which contains, (1) pairs of a system call number and its call type, and (2) a list of legitimate indirect callsites (*i.e.*, offset in a program binary). The generated metadata will be 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, which illegitimately reach system calls. The enforcement of the control-flow context is performed *only when a system call is invoked*. Our approach is different from conventional control-flow integrity (CFI) techniques, which enforce the integrity of *every (indirect) control-flow transfer*.

BASTION analyzes a control-flow graph (CFG) of an entire program and identifies all function call-level *callee→caller* relationships that reach system call callsites. To create a CFG for the whole program, BASTION merges all source code of a program into a single LLVM IR module using the `llvm-link` utility. For each system call callsite in the CFG, the BASTION compiler recursively records all callee→caller associations. The recursive analysis stops when reaching either the `main` function or an indirect function call. Our BASTION runtime monitor verifies the callee–caller relations up until the bottom of stack frame (*i.e.*, `main`), or an indirect callsite, by unwinding the stack frames.

With the call-type context and control-flow context in tandem, BASTION verifies that a control-flow reaching a system call follows (1) legitimate direct callee-caller relations and

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 4.1: BASTION library API for argument integrity. BASTION manages the shadow copy of sensitive variables (ctx_write_mem) and binds memory-backed (direct or extended) arguments and constant arguments (ctx_bind_mem_X, ctx_bind_const_X) to a specific position (X-th argument) of a system call callsite (rip).

(2) is a legitimate indirect call from a valid indirect callsite. Once the analysis is done, BASTION generates metadata for the control-flow context, which are pairs of callee and caller addresses in a program binary.

4.6.3 Analysis for Argument Integrity Context

The BASTION compiler analyzes and instruments a program to enforce argument integrity of system calls. 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 compiler instruments a program for dynamic tracking of argument variables (Section 4.6.3), and what metadata are 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 4.4 shows how BASTION enforces the argument integrity of an 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. BASTION enforces the integrity

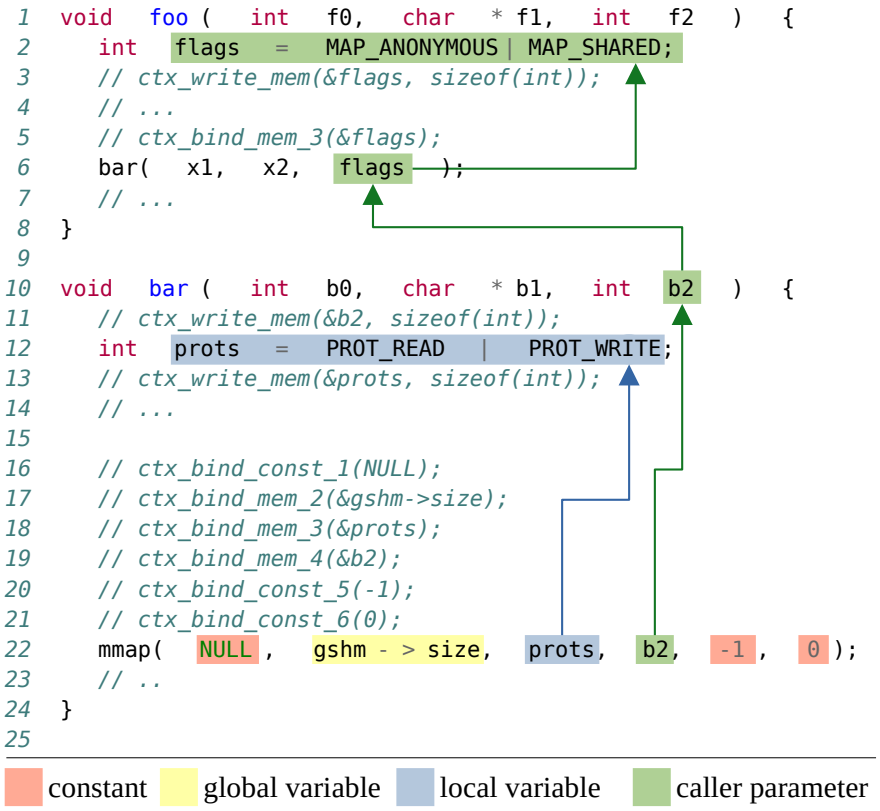


Figure 4.4: BASTION's program instrumentation for argument integrity. BASTION binds constant arguments to specified values (NULL, -1, 0) using ctx_bind_const_X. It manages the shadow copy of memory-backed arguments (gshm->size, prots, b2) using ctx_write_mem when assigned. It then binds memory-backed direct arguments using ctx_bind_mem_X before calling the mmap system call. The protection of memory-backed arguments is extended using a field-sensitive use-def analysis (size field of gshm, b2←flags) at an inter-procedural level.

of all sensitive variables.

Checking Argument Integrity

In order to verify the argument integrity of each system call, BASTION adopts data value integrity [65] for each sensitive variable. BASTION maintains the shadow copy of a sensitive variable's legitimate value in a shadow memory region and updates the shadow copy whenever a sensitive variable is updated legitimately. Before calling a system call, BASTION binds each system call argument to a certain argument position of a system call in a specific callsite

so the BASTION runtime monitor can check argument integrity using the BASTION-maintained shadow copy of the respective sensitive variables.

The BASTION runtime library provides APIs shown in Table 4.1. 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, -1, and 0) does not need to have a shadow copy, because its legitimate value is determined at the compilation time. Hence ctx_write_mem is only used for memory-backed sensitive variables (*e.g.*, gshm->size, prots, b2, and flags in Figure 4.4). 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 X-th argument of the associated system call to a constant value c. Note that binding is applied to both system call callsites as well as callsites passing sensitive variables (*e.g.*, bar() callsite in Line 6 of Figure 4.4). We note that it is not necessary to instrument if an argument is a direct or extended argument. That is because such distinction is position-specific. For example, the first argument of execve is an extended argument, for which the BASTION runtime monitor needs to investigate not only the pointer value but also pointee memory contents.

Instrumenting Sensitive Variables

The BASTION compiler identifies not only *system call arguments* but also their *data-dependent variables*, which can affect the value of a system call argument. These are collectively named *sensitive variables*. At the high level, the BASTION compiler performs field-sensitive, inter-procedural analysis to identify all data-dependent variables. It does this by traversing the use-def chain of system call arguments and analyzing any writes to a struct field used as a system call argument.

To identify all sensitive variables, the BASTION compiler 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, following use-def chains to find out any other variables that are 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 struct in Figure 4.4), that write is added to the sensitive variables, and repeats the second and third steps until no new sensitive variables are found. Note that our field-sensitive analysis is also effective for tracking sensitive global and heap variables, which can be updated off the call chain to a target system call.

BASTION first follows and instruments the callsite's system call argument variables via intra-procedural analysis (*i.e.*, within a current function). If an argument variable is updated by a caller function's parameter (*e.g.*, b2←flags in Figure 4.4), 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 4.4). This process is done recursively until an origin for the deepest use-def chain variable is found (*e.g.*, constant assignment, local variable, global variable).

Once all sensitive variables are identified, BASTION instruments ctx_write_mem right after any store instruction to a memory-backed sensitive variable to maintain its shadow copy up-to-date. Before calling a system call, BASTION instruments ctx_bind_mem_X ctx_bind_const_X to bind an argument to a specific argument position X at a certain callsite.

BASTION-compiler Generated Metadata

Once the instrumentation is done, BASTION compiler generates metadata so that our monitor can efficiently enforce the argument integrity context when reaching a sensitive system call. The metadata specifies an entry for each callsite. Each callsite entry includes the callsite file offset and the argument types (*i.e.*, constant vs. memory-backed 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 runtime monitor enforces all three of our proposed system call contexts at runtime. It is built as a separate process to ensure an attacker cannot reach or disable our BASTION enforcement mechanism from within a BASTION-protected application. We now describe how the monitor is initialized and how each context is enforced.

4.7.1 Initializing BASTION Monitor

Loading metadata. The BASTION monitor is invoked with a target application binary path and any additional arguments the application desires. It reads ELF [88] and DWARF [95] information of the application and its linked libraries to retrieve symbol addresses. It then loads BASTION context metadata of the application into the monitor's own memory.

Launching a BASTION-protected application. After loading an application's BASTION

metadata, BASTION performs a fork to spawn a child process where the child process runs the BASTION-instrumented application binary 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. Then BASTION initializes seccomp [72, 75] to trap a system call invocation of the child process and ptrace [89] to access the application's state (including the shared shadow region attached to the application's virtual address space). In this way, when a (sensitive) system call invocation attempt is made by the application, it will signal the BASTION monitor to perform context integrity verification by poking the application's execution state, before allowing the system call to be executed. While no system calls are invoked, BASTION monitor rests in an idle state via waitpid until the next sensitive system call is invoked. Once both target application and BASTION monitor processes are synchronized, the BASTION monitor allows the application to proceed and begins monitoring.

Trapping a system call invocation. The BASTION monitor initializes seccomp-BPF to trap on the application's sensitive system call invocations. It allows the use of non-sensitive system calls via SECCOMP_RET_ALLOW so that invoking a non-sensitive system call does not trap to the monitor. For sensitive system calls, the BASTION monitor disables any not-callable system calls using SECCOMP_RET_KILL. For directly- and indirectly-callable system calls, it specifies a SECCOMP_RET_TRACE policy so that such system calls are interrupted until BASTION monitor verification is complete. Note that a child process or a thread spawned by a BASTION-protected application has the same seccomp policy as its parent process and they are protected by the same BASTION monitor process.

Accessing application state and shadow memory. The BASTION monitor needs to retrieve an application's runtime information – such as register values, stack, and heap memory – and its shadow memory to verify the integrity of the system call contexts before

executing a sensitive system call. The BASTION monitor retrieves the current register values and system call number via using ptrace. It uses the process_vm_readv [73] system call to access arbitrary memory in the application's address space, including the stack, heap, and shadow memory region.

As briefly discussed, shadow memory is initialized when a BASTION-protected application is launched and it 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. In both cases, the key to access the hash table data is an address; the address of the sensitive variable and the callsite address are used to access its shadow copy and argument binding information, respectively. Note that BASTION's shadow memory region relies on sparse address space support of the underlying OS like metadata store designs in prior studies [65, 84].

4.7.2 Enforcing Call-Type Context

For the call-type context, the BASTION monitor retrieves the system call number and the program counter (rip) where the system call is invoked using ptrace. BASTION monitor uses the rip value to retrieve the call type of the callsite. For example, the BASTION monitor decodes the call instruction at Line 22 in Figure 4.4 using rip to determine if the mmap call was made direct or indirect. Then it verifies that the call type being used is allowed for the given system call by comparing the call type to the metadata. If the call type is allowed, BASTION continues to the next context check. Otherwise, BASTION monitor assumes this is an attack attempt and immediately kills the protected application and gracefully ends the BASTION monitor process.

4.7.3 Enforcing Control-Flow Context

For the control-flow context, the BASTION monitor retrieves a copy of the current stack trace when a system call attempt is made and verifies whether the current call stack follows 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 (*i.e.*, the current stack frame) in the CFG metadata. For example, in Figure 4.4, 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. If a mismatch in a control-flow transition is found, BASTION monitor assumes that a control-flow hijacking attack has been attempted to illegitimately reach this system call, and correspondingly kills the application, as done for the Call-Type context.

4.7.4 Enforcing Argument Integrity Context

For the argument integrity context, the BASTION monitor verifies the integrity of all sensitive variables in the current call stack. Take Figure 4.4 as an example. At function bar()'s stack frame, BASTION monitor verifies all bound constant variables (*i.e.*, NULL, -1, and 0) and memory-backed variables (*i.e.*, gshm->size, prots, and 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 appropriate argument integrity context metadata that reports the type of each argument (constant vs. memory-backed arguments) to know how to verify it. Particularly for non-system call callsites, BASTION also refers to the argument integrity context metadata to determine which arguments need to be verified. For each callsite's sensitive argument, the

BASTION monitor compares the actual argument value to the BASTION-traced value retrieved from the shadow memory region. BASTION iteratively traverses all relevant function frames currently on the stack. If the values match, the argument values are legitimate and the system call can proceed to be processed by the kernel. Otherwise, the BASTION monitor kills the application and concludes runtime monitoring.

4.8 Preliminary Evaluation

In this section, we evaluate how effective BASTION is when employing the collective set of system call contexts we propose. We measure the performance overhead of BASTION with real-world applications, such as the NGINX web server [2]. Our evaluation includes partitioning the individual costs of each BASTION proposed system call context, including the enforcement of Call-Type, Control-Flow, and Argument Integrity contexts to examine the cost-benefit analysis for each.

4.8.1 Evaluation Methodology

Evaluation Setup. We ran all experiments on a 24-core (48-hardware threads) machine equipped with two Intel Xeon Silver 4116 CPUs (2.10 GHz) and 128 GB DRAM. All benchmarks were compiled with the BASTION LLVM compiler. For performance results, we report the average over four runs.

Applications. We chose to evaluate NGINX as this application is widely deployed and as such, is often victim to attack. This application is a sufficient litmus test to ascertain the effectiveness of BASTION as it is both an I/O intensive and system call heavy application.

To test the throughput of NGINX, we use wrk [50], an HTTP benchmarking tool. wrk sends

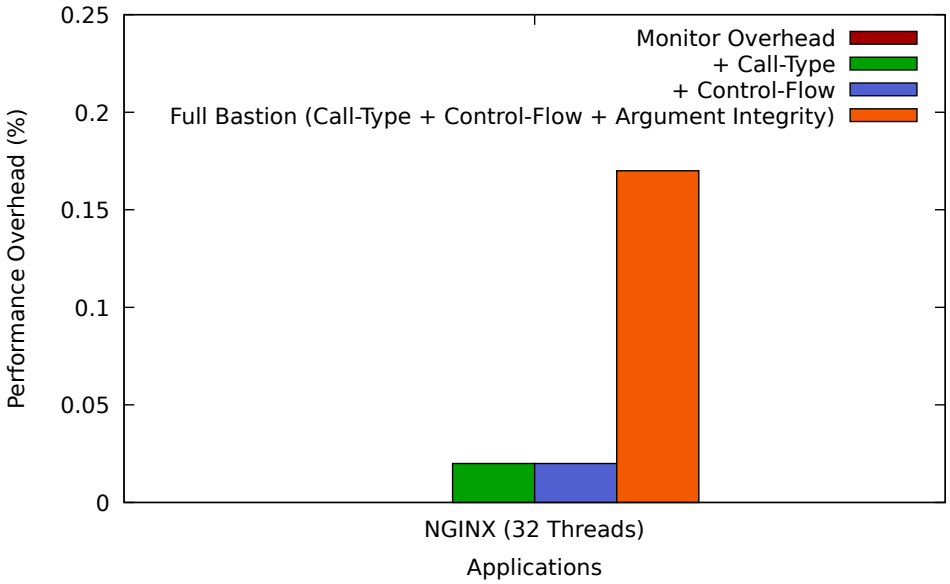


Figure 4.5: Performance breakdown for each of BASTION's system call contexts.

concurrent HTTP requests to a web server to measure the throughput. We place the wrk client on a different machine still on the same local network as the NGINX webserver.

4.8.2 Performance Evaluation

NGINX. We first examine BASTION's impact on high I/O based applications via running wrk [50] with NGINX [2]. We measure its throughput for a 20-second run while varying the number of NGINX workers. NGINX is configured to handle a maximum of 1,024 connections per processor, and 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.

For all NGINX worker counts (1, 4, 8, 16, & 32) evaluated, the runtime overhead while applying full BASTION protection (all three context checks enabled, Call-Type, Control-Flow, & Argument Integrity) was never more than %0.17 degradation compared to an unprotected NGINX baseline. Figure 4.5 specificially shows the breakdown of incrementally adding each

system call context for runtime enforcement for NGINX (16 workers and 32 workers). For NGINX, the monitor, Call-Type, and Control-Flow contexts contribute less than 0.02% overhead for each component.

Examining the code design of NGINX, it is apparent that NGINX specifically utilizes a vast majority of sensitive system calls (*e.g.*, mprotect, mmap) during its' initialization phase while barely using any sensitive system calls when in its idle state or processing requests. This results in BASTION rarely being triggered during actual runtime to verify system call usage integrity. 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 the minimum and maximum Control-Flow call-depths encountered, respectively.

Summary The performance evaluation of NGINX confirm our insights of sensitive system call usage patterns to advantageously leverage them for efficient enforcement of legitimate system call usage during runtime. BASTION shows these contexts are sufficient both as a standalone defense as in BASTION, and as a supplemental tool to further harden modern defense techniques since our evaluation of these contexts gives evidence these system call contexts can be combined with existing defense techniques for a low performance cost. Enforcement of the Call-Type context has a negligible performance impact and provides the first step of enhancing system call filtering constraints by adding the additional context check of how a system call is invoked by the application during runtime. Moreover, since Control-Flow context enforcement is designed as a much more narrow type of CFI, BASTION is able to omit many unnecessary intermediary integrity checks. Even if control-flow is illegitimately altered in non-system call related stack traces, the inherent security invariants of our proposed contexts for system call integrity guarantee that an attack will be defeated once it attempts to use a system call in order to try to complete the attack. Finally, Argument Integrity context enforcement is still the most expensive property, even for BASTION.

However, BASTION greatly reduces the cost of runtime tracing while still hardening the security for legitimate system call usage by defending the system call argument context.

4.9 Related Work

Our three proposed system call contexts are broadly influenced by several modern defense techniques. In the following, we discuss these related works and how BASTION differs.

Basic System Call Filtering System call filtering is a technique that proactively focuses on securing legitimate system call usage. However, even for this approach, the majority of filtering techniques are one dimensional, only offering to guard system calls through whitelisting and rudimentary, coarse-grained argument enforcement. Compare this to BASTION, which proactively protects three different aspects of legitimate system call usage and enforces per-argument per-callsite value integrity. Static-based filtering techniques aim to make the process of employing coarse-grained filters more user-friendly through the use of libraries (libseccomp [66]), automation (sysfilter [32]), or portability across platforms and architectures (ABHAYA [103]). Dynamic filtering techniques instead aim to improve the overall soundness of filtering through automated testing [67, 135] or dynamic profiling, as in Confine [46]. In contrast, BASTION provides automatic analysis & instrumentation, metadata generation, and is sound without requiring any annotation or test-cases from a user.

Temporal System Call Filtering Temporal Specialization [47] and SPEAKER [87] leverage a temporal context, where specific system calls are only enabled for certain parts of a user program's phase of execution. These are approaches that tighten system call usage. Similarly, SHARD [5] leverages temporal context to perform kernel code specialization, depending on the host application and system call invoked. Using a temporal context is valid, but not sufficient to defend against advanced code-reuse attacks. BASTION uses Call-Type,

Control-Flow, and Argument Integrity collectively to ensure system calls are called correctly, and have verfied input arguments.

Attack Surface Reduction Techniques such as CHISEL [56], Nibbler [6], Piece-wise [111], RAZOR [110], and TRIMMER [118] all use various forms of debloating to identify and remove unused program code. The two key limitations of debloating techniques are that they do not consider system call context to provide additional security and they are heavy-handed, fragile methodologies that require recompilation for any configuration change. BASTION needs to only be analyzed and compiled once regardless of configuration changes, and leverages context for legitimate system call usage.

Control Flow Integrity Control Flow Integrity (CFI) aims to restrict the control-flow of an application to only legitimate execution paths and legal indirect callsite targets. Unlike CFI-style techniques, BASTION only protects callsites targeting system calls. Moreover, BASTION verifies the integrity of system calls invoked via both direct and indirect calls. Additionally, CFI-style techniques do not offer any value integrity mechanism for system call arguments; BASTION protects all sensitive variables such that it can additionally defend against non-control data attacks that leverage system calls.

Data Flow Integrity Data integrity, such as DFI [19], defends against both control and non-control data attacks by tracing all data and only allowing legitimate data updates. Compared to DFI, which is applied application wide for all variables, BASTION only enforces argument integrity for *sensitive variables*. Given that BASTION's entire premise revolves around system calls, we show that it is sufficient to only protect system call arguments, when other system call contexts are checked in unison to block attacks leveraging system calls.

Saffire [98], performs function specialization by creating specialized restricted argument interfaces exclusive to each specific callsite in an application for a predefined a list of security-

critical functions. Specifically, for dynamic argument values, Saffire constrains those arguments to only allow certain value sets or value ranges. However, Saffire's data flow integrity check only cross references against the most recent legitimately marked update of the dynamic argument in question. Compare this to BASTION, which enforces full value integrity of all system call arguments and *sensitive variables* from their origin. Also, Saffire experiences similar problems as CFI-style techniques, where for undeterminable arguments, it sets up an equivalence class of valid argument values. This opens up the possibility that an attacker desired value could be within that equivalence class.

4.10 Summary

This work is built on the insight that regardless of attack complexity or end goal, a majority of attacks *must* leverage system calls in order to complete their attack. 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. The Call-Type context forms tighter system call filtering. The Control-Flow context is a narrower form of CFI, specific to system calls. Moreover, the Argument Integrity context hardens legitimate system call usage by ensuring all arguments are genuine.

Evaluating the performance impact of BASTION using NGINX, we demonstrate a low runtime overhead of 0.17%. This shows BASTION is a practical defense on its own to block an entire attack class that leverages system calls, and that the newly introduced system call contexts can seamlessly complement existing modern defenses.

Chapter 5

Ongoing Research & Future Work

In this chapter, proposed post-preliminary future work is described. The MARDU defense framework has already been fully implemented and evaluated to demonstrate its merits and contributions towards re-randomization-based defense techniques. For BASTION, this thesis proposal has presented its motivation, design, as well as a preliminary performance evaluation. In order to be considered a comprehensive defense framework, BASTION requires several additional milestones to be completed. Section 5.1 explains the needed code support to consider BASTION a viable defense prototype. Section 5.2 then proposes a security study for BASTION to confirm the effectiveness of the three proposed specialized system call contexts it employs. Finally, Section 5.3 proposes a performance study that evaluates a diversified set of applications and benchmarks.

5.1 Completed BASTION Prototype

Currently, BASTION has a priliminary bare-bones prototype in place. Nonetheless, this prototype has allowed us to get a glimpse at promising evaluation results. We propose to

finish the implementation details necessary to support additional real-world applications. This includes hardening the the BASTION LLVM compiler to support processing of more complex code patterns commonly present in real-world applications as in those applicatins mentioned below. Additionally, automation is needed for portions of the analysis and the process of generating various BASTION specific metadata needed for runtime.

5.2 Comprehensive Security Study of BASTION

Defense techniques traditionally provide a security evaluation to show the beneficial properties that can be gained when employing said defense technique. Common metrics to show security effectiveness include statistics such as reduced attack surface, by means of eliminated unused code functions or code gadgets [6, 56, 83, 110, 111], disabled system calls [32, 46, 47, 75, 103], or reduction of code pointer equivalence class size [60, 77]. Additionally, significant bodies of work carry out trials against various attack scenarios to also test a defenses security guarantees, using either originally formulated or procured attacks from works that specifically publicize new attack models [114, 134]. Thus, we propose to evaluate BASTION in this way to provide evidence of upholding its proposed security guarantees and invariants.

5.3 Comprehensive Performance Study of BASTION

Currently as described in Section 4.8, BASTION has only been evaluated using the NGINX webserver. We propose studying additional applications and benchmarks to cover other types of typical computation. BASTION will choose a mix of real-world applications that are system call instensive and compute-intensive to investigate workload patterns and differences.

Possible candidates for this performance study include:

- **SQLite** [123] – SQLite is a widely deployed in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. Well known users of SQLite include Adobe, Apple, Facebook, Google, Microsoft, and Mozilla Firefox.
- **vsftpd** [3] – vsftpd is a popular FTP server for UNIX systems, including Linux. It is the default FTP server in the Ubuntu, CentOS, Fedora, NimbleX, Slackware and RHEL Linux distributions.
- **nBench** [94] – This is a benchmark suite that has been ported to Linux/Unix from BYTE's Native Mode Benchmarks. It is designed to test the capabilities of a system's CPU, FPU, and memory system.
- **CoreMark** [37] – This is another standard benchmark designed specifically to measure the performance of microcontrollers (MCUs) and central processing units (CPUs). This is a compute-intensive benchmark created by the Embedded Microprocessor Benchmark Consortium (EEMBC).
- **SPEC CPU2006** [55] [1] – The SPEC benchmark suite is a standard benchmark set for system software research. These include both integer and floating-point computations that focus on providing realistic and varied real-world computations, including compilation, scripting, image compression, *etc.*.

By obtaining a comprehensive performance evaluation of BASTION, a profile for each of the three proposed system call contexts can be deduced which could reveal new trends and insights in system call usage patterns that could be leveraged further in future system call protection research.

[1] An initial source code survey of SPEC shows very limited system call usage (*e.g.*, system calls only used at program initialization to allocate memory for datasets and information) therefore this may not be a good performance evaluation candidate

Chapter 6

Conclusions

This thesis proposal presents two interrelated research projects that address the viability of making practical and strong defense mechanisms. Both MARDU and BASTION show implementations of practical defense mechanisms that do not sacrifice security guarantees and are comparable to modern defense approaches. Completing the second research thrust with BASTION will add evidence that system calls are an applicable attack vector code component that has value in being defended by this thesis's proposed defense.

This proposal's prior research, MARDU, investigates and designs a solution to support randomization of shared code that also exhibits minimal performance penalties. MARDU focuses on the protection of code gadgets against prevalant ROP attacks and implements a full defense framework that is capable of addressing those attacks. MARDU introduces a novel contribution of figuring out how to achieve scalable code sharing with diversification in applications (especially shared library code). This contribution removed redundant computation like tracking, patching, and greatly reduced inherent memory overhead prevalent in recent randomization defenses. By randomizing code only when attacker activities are detected (*e.g.*, crashing workers) and leveraging trampolines with a secure memory region,

code and code pointer leakage can be prevented. MARDU has been evaluated with both standard performance benchmarks (SPEC CPU2006) as well as real-world applications such as NGINX webserver. When MARDU is deployed, performance degradation was evaluated to not be more than 5.5% and 4.4%, respectively. Even when under constant attack, MARDU only adds no more than 15% additional overhead for the most complex applications. Finally, MARDU defended against all attacks considered by load-time and continuous re-randomization techniques. These attacks included JIT-ROP attacks, code inference attacks, low-profile timing attacks, and code pointer offsetting attacks.

This thesis also described the design for contexts specialized toward protection and legitimate usage of system calls in user applications. Current defenses are still impractical and exhibit too many tradeoffs in performance or resource consumption leading to very few proposed techniques being adopted mainstream. Additionally, recent defenses that do attempt to address the exposed nature of system calls are incomplete or naive. This thesis proposal introduces BASTION in order to address these gaps in current defenses. BASTION specifically proposes three contexts to enforce legitimate system call usage. First, BASTION makes sure that for those system calls that are used by an application, they are call with the expected calling convention either with an direct or indirect call. Second, BASTION ensures that control-flow paths reaching a system call are legitimate by assessing a narrower form of CFI. Finally, BASTION verifies value integrity for all system call arguments and all data-dependent variables such that all system call arguments are genuine. Although BASTION is still in development, a preliminary evaluation shows that promising results. Specifically, BASTION has been only evaluated with a real-world application, NGINX. These results show that BASTION has an overhead of less than 0.17% for NGINX. BASTION has a substantially smaller performance footprint than other conventional defense approaches like CFI and re-randomization, however there is no easily identified counterpart to compare with.

BASTION takes a significantly different angle in its protection approach. That being said, a comprehensive security study is needed to confirm our design's security guarantee's. Additionally, a supplementary performance evaluation can further solidify this proposal's initial reported findings. BASTION takes on a new approach to preventing attacks, by going after a probable lynchpin component, system calls. By solely ensuring that this component is adequately protected, all attack clases depending on system call usage can be effectively nullified.

Bibliography

[1] musl libc, 2022. https://wiki.musl-libc.org/.

[2] NGINX Web Server, 2022. nginx.org/.

[3] vsftpd, 2022. https://security.appspot.com/vsftpd.html.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] EEMBC. CoreMark - An EEMBC Benchmark, 2022. https://www.eembc.org/coremark.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. doi: 10.1145/1186736.1186737. URL http://doi.acm.org/10.1145/1186736.1186737.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

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

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

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

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

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

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

[74] Juszkiewicz, Marcin. Linux kernel system calls for all architectures, March 2022. https://marcin.juszkiewicz.com.pl/download/tables/syscalls.html.

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

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

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

[86] 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.

[87] 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.

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

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

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] Uwe Mayer. Linux/Unix nbench, 2017. https://www.math.utah.edu/~mayer/linux/bmark.html.

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

[96] 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.

[97] 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.

[98] 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.

[99] 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.

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

[101] 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.

[102] 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.

[103] 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.

[104] 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.

[105] 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.

[106] 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.

[107] 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.

[108] 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.

[109] 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.

[110] 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.

[111] 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.

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

[113] 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.

[114] 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.

[115] 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.

[116] 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.

[117] 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.

[118] 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.

[119] 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.

[120] 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.

[121] 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.

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

[123] SQLite. SQLite, 2022. https://www.sqlite.org/index.html.

[124] 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.

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

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

[127] 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.

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

[129] 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.

[130] 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.

[131] 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.

[132] 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.

[133] 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.

[134] 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.

[135] 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.

[136] 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.

[137] 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.

[138] 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.

[139] 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.

[140] 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.

[141] 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.

[142] 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 Con-*

ference on Architectural Support for Programming Languages and Operating Systems, pages 133–147, 2020.

[143] 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.

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

[145] 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.

[146] 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.