UNIVERSITY OF CALIFORNIA, SAN DIEGO

On the Effectiveness of Heterogeneous-ISA Program State Relocation against Return-Oriented Programming

A thesis submitted in partial satisfaction of the
requirements for the degree of Master of Science

in

Computer Science

by

Sriskanda Shamasunder

Committee in charge:

Professor Dean Tullsen, Chair
Professor Michael Taylor
Professor Stefan Savage

2015

The Thesis of Sriskanda Shamasunder is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2015

DEDICATION

Dedicated to my parents,
Rajeshwari and Shamasunder,
for my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude for the help and support they have provided throughout my graduate studies to the following people:

Professor Dean Tullsen, for his support as the chair of my committee, for his guidance and advice throughout my graduate studies as my graduate adviser, and for giving me the privilege of working with him for two years.

Ashish Venkat, for his invaluable help over the last two years as my mentor. Without his guidance I would not have not known where to start or how to finish.

Sam and Andreas, for being such welcoming and entertaining lab mates.

And most importantly, Pooja Suri, for being my pillar of support and standing by me through thick and thin over the course of my graduate studies. I couldn't have done it without you.

Chapters 3, 4, 5, 6, 7, and 9 are in part currently being prepared for submission for publication of the material. Venkat, Ashish; Shamasunder, Sriskanda; Tullsen, Dean; Shacham, Hovav. The thesis author was one of the primary investigators and co-author of this material.

VITA

2006–2010	Bachelor of Engineering in Computer Science, Sir M Visvesvaraya Institute of Technology, Bangalore
2008–2010	Environment Specialist, Intel Technologies India Pvt Ltd, Bangalore
2010–2013	Senior Member Technical, D. E. Shaw & Co, Hyderabad
2014	Production Engineer Intern, Facebook Ltd., Menlo Park
2014–2015	Teaching Assistant, University of California, San Diego
2014–2015	Research Assistant, University of California, San Diego
2015	Master of Science in Computer Science, University of California, San Diego

FIELDS OF STUDY

Studies in Computer Architecture
Professor Dean Tullsen, University of California, San Diego

ABSTRACT OF THE THESIS

On the Effectiveness of Heterogeneous-ISA Program State Relocation against Return-Oriented Programming

by

Sriskanda Shamasunder

Master of Science in Computer Science

University of California, San Diego, 2015

Professor Dean Tullsen, Chair

As computer software grow larger in size and complexity, there is an ever increasing concern over security. In an age where software controls almost everything, from the cars we drive to the airplanes we fly in, this concern is valid now more than ever. Attackers are evolving new ways to exploit vulnerabilities in software everyday, while the computer security community struggles to keep up. One of the most prominent of these attack methods is code reuse attacks - specifically return-oriented programming and its variants.

Traditionally, defense techniques have mostly either been at the hardware level

or in the software layer. While these defenses have their own strengths and weaknesses, a layer of abstraction that has mostly been unexplored is the architecture. Computer architecture lies at the boundary of hardware and software, where we can harness the strengths of both layers. This work explores the potential security benefit that we can extract from decoupling the architectural state that the system presents to the software, from the micro-architectural state it maintains in hardware.

Recent research has shown the potential for heterogeneous-ISA chip multiprocessors to provide both performance and energy benefits. We propose Heterogeneous-ISA Program State Relocation, an architecture based on heterogeneous-ISA computing that randomizes the ISA a program executes on, and couple that with a defense mechanism that dynamically randomizes the program state. We describe the proposed architecture, our implementation of it, and perform a thorough evaluation of its potential as an effective defense technique.

Chapter 1

Introduction

Computers and software have become ubiquitous in almost every aspect of our lives. We have computer systems that control space shuttles and satellites, and software on our phones that control our homes. But one thing for sure is that we are still a long way from proclaiming that our systems and software are completely secure from attacks, either from attackers with a malicious intent, or from governments that are increasingly inquisitive. While there have been several attempts at building better tools and developing better processes to avoid introducing bugs in software that lead to vulnerabilities, the vast majority of the code executed today was still written decades ago [44]. This has left the security of our software in the hands of two kinds of people, one trying to find newer and more cunning ways to attack and exploit these vulnerabilities, and the other trying to stop these attackers by building stronger defenses to patch these vulnerabilities and secure our software.

Code reuse attacks exploit vulnerabilities in software to hijack control flow and execute arbitrary code as directed by the attacker to perform malicious computation. While there have been several techniques propounded to defend against these attacks, both software and hardware based, they all come with their own caveats of either heavy performance degradation or daunting complexity.

The unique position of computer architects as the designers of the interface

between hardware software puts them in a position to leverage both hardware and software to address the issue of security. By decoupling the state of the system as observed by an attacker, or even the program itself, and the actual micro-architectural state of the executing process, we can insulate software from causing unintended external effects or from being affected by them. This decoupling of architecture and micro-architecture forms the theme for this thesis where we present and evaluate an architectural defense technique to defend against code reuse attacks.

One of the main challenges in developing a new defense technique is validating its effectiveness. The lack of standard metrics that define what constitutes a good defense makes it difficult to quantify the security benefits obtained from a defense mechanism. This is especially true in the case of a randomization based defense since theoretical proofs cannot accurately capture the effects of a practical implementation. Experimentally evaluating a defense against only a handful of known attacks is insufficient to deem its capacity to prevent new ones, but at the same time simulating all possible scenarios quickly becomes intractable. In this work we focus on metrics that can effectively represent the vulnerability of a system and develop methods to make exhaustive experimental evaluation computationally feasible.

We present Heterogeneous-ISA Program State Relocation, a defense technique that randomizes the program state and underlying ISA in ways that allow legitimate execution to run undeterred but make it next to impossible for an attacker to perform malicious computation that deviates from the intended control flow of the program. Heterogeneous-ISA Program State Relocation as a secure architecture is a joint work with Ashish Venkat where we propose the underlying architecture, describe its implementation, and evaluate its effectiveness.

The thesis consists of two main sections: The first half of the thesis (Chapters 4 and 5) is joint work that is presented to provide the reader with requisite background on

Heterogeneous-ISA Program State Relocation. The second half (Chapters 7, 8 and 9) presents the contribution of this work - the evaluation of the proposed architecture for its effectiveness as a defense technique.

Chapter 2

Overview

Code reuse attacks such as return-oriented programming (ROP) [48, 51] are a class of exploits that have become an attacker's weapon of choice against today's systems. After nearly a decade, security researchers are yet to find a way to stop these attacks completely. The strength of code reuse attacks lies in the fact that they recycle the victim program's code to piece together malicious code. This makes it difficult for conventional attack detection/prevention mechanisms to distinguish malicious computation from legitimate execution.

There have been several defenses proposed in literature against ROP, and some even commercially deployed [10], but almost all of them have failed to provide a comprehensive defense either because of prohibitively high performance costs, impractical complexity or the inability to keep up with rapidly evolving attackers. For example, Control Flow Integrity (CFI) [9, 40, 8, 55] based techniques, as comprehensive as they are, have a high overhead in terms of performance and source code information required to implement them. Some randomization based techniques such as Address Space Layout Randomization (ASLR) [57] have been shown to be ineffective against an attacker with the time and resources to brute force it [52]. Most other compile-time or load-time randomization techniques have been undermined by a new variant of ROP called Just-in-Time Code Reuse (JIT-ROP) [54] attacks. What then is needed is a defense mechanism

that is baked into the architecture of the machine itself and is capable of protecting a system against code reuse attacks with minimal impact on performance.

2.1 Potential of Heterogeneous Chip Multiprocessors

Heterogeneous chip multiprocessors employ CPU cores of different organization or size that offer varying degrees of micro-architectural complexity [34, 7, 27, 28] and/or core specialization [4, 5, 6, 38]. Owing to their high performance and execution efficiency, these architectures have been showcased, for example, as promising candidates towards achieving energy proportionality in large data-centers [13, 36, 58]. While early work on on-chip heterogeneity[33, 32] restricted cores to implement a single instruction set architecture (ISA), recent findings [25, 59] indicate that a heterogeneous-ISA chip multiprocessor (CMP) is not only a viable option, but has greater potential both in terms of performance and energy efficiency.

A heterogeneous-ISA CMP synergistically complements architectural heterogeneity with micro-architectural heterogeneity, and allows an application to dynamically identify the ISA of its preference and migrate execution at any given point of time. By migrating execution not just when it is beneficial for performance, but also when there is the threat of an attack, heterogeneous-ISA computing can effectively remove one of the last underlying assumptions of an attacker - the ISA. In this work, we leverage this architecture to demonstrate significant new security benefits, and in particular, showcase its ability to defend against an evasive class of buffer overflow exploits called Return-oriented Programming [48, 51].

2.2 Randomizing Program State

Every program, even a return-oriented program, requires a certain amount of program state in the form of registers and memory locations to perform meaningful

computation. Randomization techniques have until now primarily targeted the location and form of program code in order to thwart code reuse attacks. One aspect of a program's execution that is presumed by attackers to be guaranteed, is the program state.

This program state can be leveraged as a potential source of security by randomizing the it in such a way that legitimate execution can continue unhindered, but any malicious computation that deviates from the expected control flow most likely fails. We employ a dynamic binary translator (DBT) to transform program code on-the-fly and relocate program state into random, attacker-unknown locations. The use of a DBT ensures that every piece of code executed, either from the program or from a dynamically linked library, is randomized and hence leaves no room for vulnerabilities.

2.3 Putting it all together

Heterogeneous-ISA Program State Relocation (PSR) is an architecture that we propose as a solution to this problem. The architecture incorporates two strong orthogonal defense techniques - Instruction Set Randomization and Program State Relocation. Instruction Set Randomization probabilistically migrates an executing process between cores of different ISAs, and Program State Relocation randomizes the program state at a run-time, thereby rendering the attacker's knowledge of the victim's binary obsolete.

The two most important concerns regarding any defense technique are its impact on performance, and its effectiveness at defending against attacks. Any defense mechanism that adversely impinges on the performance of the system, be it in execution time, demand for resources, or in usability, is deemed impractical in a world that values performance over security. At the same time, a lightweight defense mechanism that covers only a subset of attacks is ineffective. Therefore it is imperative for any proposed defense techniques to be thoroughly evaluated against these two metrics.

The scope of this work is to understand what constitutes an effective defense,

define metrics to measure effectiveness in a tangible manner, and finally evaluate Heterogeneous-ISA Program State Relocation. To this end, we evaluate our proposed architecture against a slew of attack techniques such as classic ROP, brute forced ROP and Just-In-Time ROP attacks. Since it is non-trivial to quantify the effectiveness of a defense technique, especially one based heavily on randomization to such a degree, we define a set of metrics that best represent the vulnerability of a system without and in the presence of Heterogeneous-ISA PSR. We conduct a series of experiments to measure its effectiveness on synthetic benchmarks and present our findings.

The thesis is organized as follows: Chapter 3 provides the necessary background on return-oriented programming, current defenses against ROP, and heterogeneous chip multiprocessing. Chapter 4 describes the proposed architecture of Heterogeneous-ISA PSR while Chapter 5 presents a brief discussion of its implementation. Chapter 6 describes the threat model we assume in our study. Chapter 7 describes the metrics and models used for evaluating PSR. Chapter 8 describes the methodology used in conducting the experiments. Chapter 9 presents the results from our experiments and discusses inferences from them. Chapter 10 concludes the work with a note on future work.

Chapter 3

Background and Related Work

3.1 Code Reuse Attacks

Buffer overflows are one of the most common exploits in software even today. Buffer overflow exploits are made possible by vulnerabilities in software that result from buggy code that fails to perform the appropriate bounds checking. Attackers have exploited these vulnerabilities for decades to abuse systems and software, but the frequency, complexity and ingenuity of these attacks has been ever increasing.

The first buffer overflow exploits such as Stack Smashing [42] overwrote the return address on the stack to divert control flow to attacker injected code, also on the stack. Several defense techniques were proposed to mitigate such attacks, both software-based [11, 21] and hardware-based [39, 43]. One of the most prevalent mitigation techniques is $W \oplus X$, which essentially marks code pages as either writable or executable, but never both. This has been deployed widely on both Windows, as Data Execution Prevention (DEP) [10], and on Linux through the PaX patch [56], using the NX bit supported by most modern CPUs. While this deterred code injection attacks, it led to the evolution of a whole new breed of attacks based on code re-use instead.

3.1.1 Return-to-libc

Code reuse attacks exploit vulnerabilities in systems to reroute control flow into existing code rather than injected code with the intent of performing malicious computation. Return-to-libc was one of the first techniques that did so by redirecting control flow to libc functions, supplying them with carefully crafted attacker supplied arguments. While such an attack can be used to bypass $W \oplus X$, the spectrum of computation that the attacker can perform is still limited by the functions available in libc. What the attackers required was a way to perform arbitrary computation.

3.1.2 Return-oriented Programming

This led to the evolution of Return-oriented Programming (ROP) [52, 51]. Return-oriented programming bridges the gap between return-to-libc and code injection by reusing snippets of code from the victim program to stitch together exploits that perform arbitrary execution. These snippets of code, called gadgets, comprise any set of instructions that end with a return. The attacker uses many such gadgets, each performing a small computation, to put together a malicious exploit. Given a sufficiently large code base, it has been proven that these gadgets can be used to perform any form of computation, in other words, return-oriented programming has been proven to be Turing complete [48].

Return-oriented programming hinges on the attacker being able to control both the instruction pointer and the stack pointer, the instruction pointer to direct control flow to the first gadget and then subsequently use the stack pointer to direct control flow to the next gadget in the chain. In this sense, the attacker uses the stack pointer as the instruction pointer during the execution of the exploit. Some forms of the attack also inject the payload on to the heap instead of the stack and modify the stack pointer itself. There have also been variants of return-oriented programming proposed that do away

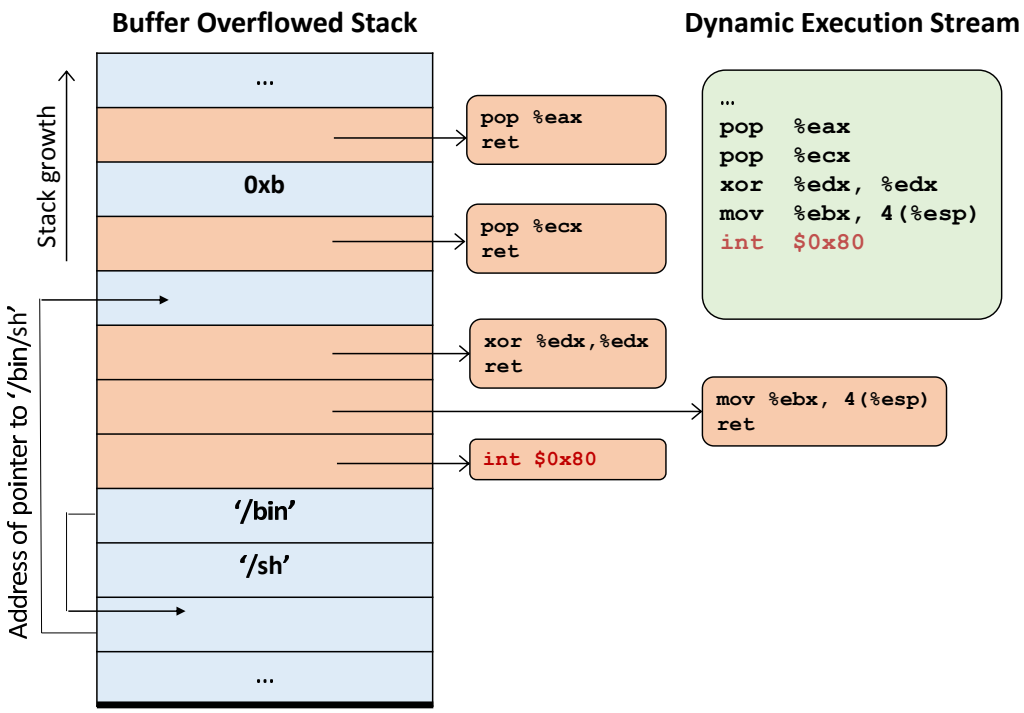

Figure 3.1. Return-oriented Programming

with the requirement of gadgets to end with a *ret*. Jump-oriented programming [19] uses indirect jumps in place of returns to subvert control flow, with the help of a "dispatcher gadget" that is responsible for executing a chain of gadgets. While the original form of return-oriented programming was proposed on x86 machines, since the unaligned instruction memory access on x86 made it easier to find gadgets, ROP has been shown to be equally effective on ARM as well [31]. Figure 3.1 depicts the workings of a return-oriented programming based exploit.

One of the challenges in constructing a ROP exploit is in finding the right set of gadgets in a given binary, and constructing a precise payload that makes use of these gadgets. While the initial ROP exploits were constructed by hand, several automatic ROP payload compilers have been introduced since then [50, 3, 2]. These automated tools drastically speed up the process of gadget discovery, making ROP one of the most damaging attack methods available today.

3.2 Defenses against Code Reuse Attacks

There have been several defense techniques that have proposed in literature to mitigate ROP, and some have even been commercially deployed. These defense techniques can broadly be classified into two categories:

3.2.1 Control Flow Integrity

The success of code reuse attacks is limited by their ability to subvert the control flow of a running program to arbitrary locations in memory. Abadi, et al. [9, 8] first formalized the idea of CFI - to constrain the execution of a program to its predetermined control flow graph (CFG) by instrumenting the program to perform checks before every indirect jump. While this technique claims to completely eliminate arbitrary control flow transfers, it suffers from a prohibitively high performance cost of as much as 45%.

Since then, a number of defenses have been proposed that attempt to provide the security of CFI while lowering the performance overhead. Compact Control Flow Integrity (CCFIR) [55], Branch Regulation [30], and Opaque CFI [40] are some defense techniques that fall under this category.

kBouncer [45], ROPGuard [26], and ROPecker [20] represent more coarse grained CFI techniques that enforce a subset of CFI constraints, such as restricting the target of return instructions to call preceded instructions (ROPGuard), or detecting any deviations by analyzing recently taken branches for signatures of a ROP attack (ROPecker and kBouncer). Relaxing constraints, while good for performance, can also reduce the effectiveness of these techniques, as demonstrated by Davi, et al [23].

3.2.2 Randomization

Classic code reuse attacks also depend heavily on knowledge of the executing program's code layout and location of useful gadgets. The second class of defense

techniques attempt to nullify this knowledge of the attacker by either randomizing the location of gadgets or transforming them to render them useless.

Address Space Layout Randomization(ASLR) introduced by PaX [57], randomizes three key areas of a program's address space (a) the main executable region (code, data, bss, and brk() controlled heap), (b) mmap() managed memory (libraries, thread local storage and all other memory mapped data), and (c) the user stack. Each region is loaded at a different randomized offset during program startup. But this granularity of randomization has been demonstrated to be insufficient against brute force attacks [52]. Since then a number of techniques have attempted to increase the entropy by reducing the granularity of randomization. Binary Stirring [60], Instruction Location Randomization [29], and Code Shredding [53] represent a class of defenses that aim to reduce the granularity, from basic blocks, to instructions, to bytes. G-Free [41], and In-Place Code Randomization [46] perform load-time transformations on the code to either eliminate gadgets or replace them with equivalent instructions that break gadgets.

While the entropy provided by these techniques against ROP attacks is compelling, they are prone to attacks such as JIT-ROP [54] that bypass load-time randomization. JIT-ROP is an attack technique where an attacker exploits a memory inference vulnerability repeatedly to scan the memory image of a process in execution, discover code pages and reconstruct them, find gadgets in them and compile an exploit payload from them, all in run-time.

Isomeron [24] harnesses software diversity and probabilistic execution by loading two versions of the program, one original and one diversified, into the address space of the program and randomly switching between the two at every function call. Our work differs from Isomeron in that we diversify code within and across the ISA, thus randomizing the architecture itself. We also perform run-time program state randomization as opposed to load-time, which provides us with a unique opportunity to thwart repeated attacks

by re-randomizing between attempts. Further, a combination of the two allows our architecture to exhibit a high degree of entropy against even very short-chain exploits.

3.3 Heterogeneous Chip Multiprocessors

The benefits of heterogeneous chip multiprocessors for power and energy efficiency have been demonstrated by Kumar, et al. [33, 32], especially when coupled with an effective scheduling policy [22]. Architectures such as ARM's big.LITTLE processor [27] and NVidia's Kal-El processor [7] have since proved their commercial viability. But these architectures restrict themselves to a single ISA to allow rapid migration of threads between cores dynamically, without any transformation.

Heterogeneous-ISA CMPs further explore architectural heterogeneity by using cores that belong to multiple ISAs. The current breed of heterogeneous-ISA CMPs, mostly FPGAs, GPUs, accelerators and MPSoCs [47, 6, 17], are very specialized and lack a common address space that allows dynamic execution migration. DeVuyst, et al. [25] laid the first foundations for general purpose heterogeneous-ISA CMPs by showing that migration cost could be reduced by an order of magnitude by utilizing shared memory in place of memory transfer. Venkat and Tullsen [59] conduct a design space exploration to find the the optimal heterogeneous-ISA CMP for general purpose mixed workloads and demonstrate that such an architecture provides both performance and energy gains for a wide range of applications. These architectures require support from compatible operating systems [12, 35] and memory consistency frameworks [37]. Commercial architectures have also since made a move towards exploring heterogeneous processors (CPU-GPU) for their power efficiency by improving programmability and portability [1].

Chapter 3, in part is currently being prepared for submission for publication of the material. Venkat, Ashish; Shamasunder, Sriskanda; Tullsen, Dean; Shacham, Hovav.

The thesis author was one of the primary investigators and co-author of this material.

Chapter 4

Heterogeneous-ISA Program State Relocation

The success of a ROP attack relies on the attacker's ability to know the state of an executing program and modify it. Randomization based defenses attempt to make the attacker's job harder, if not impossible, by increasing the entropy of a system (the number of randomizable states). In our work on Heterogeneous-ISA Program State Relocation we leverage the power of Heterogeneous-ISA execution and a new randomization technique dubbed Program State Relocation to push the entropy higher than ever by decoupling a program's execution from the underlying micro-architecture.

Architects have demonstrated both the viability and efficiency advantages of a heterogeneous-ISA CMP. These architectures maximize efficiency by allowing dynamic task migration between cores executing different ISAs, possibly between different application phases, or reacting to the changing operating conditions of the processor (e.g., thermal emergency). In this chapter, we discuss strategies to harness and re-purpose these techniques as a security defense for ROP.

4.1 Instruction Set Randomization

From a security standpoint, heterogeneous-ISA CMPs have two major advantages. First, ROP attacks are highly target-ISA dependent. An application that migrates between multiple heterogeneous-ISA cores executes instructions from different instruction sets. If a migration is forced upon execution of every ROP gadget, a successful attack would involve chaining gadgets from different ISAs, and yet produce a meaningful result (e.g., spawn a shell). Furthermore, if we make migration probabilistic, we remove the most fundamental assumption of the attacker – knowledge of what ISA the gadget will execute on. The second advantage is that execution migration in a heterogeneous-ISA CMP requires stack transformation. This especially constrains ROP gadgets to save all intermediate state in locations that are immune to run-time stack transformation (e.g., heap memory), thereby significantly reducing the attack surface.

Several fine-grained randomization techniques proposed in prior work have been shown to be broken by a malicious attack called just-in-time return-oriented programming (JIT-ROP) [54] that exploits a single leaked memory disclosure to reconstruct the entire memory image of the process, and thereby bypass all randomization. Instruction Set Randomization in a heterogeneous-ISA CMP, however, severely inhibits JIT-ROP. This is because the decision to migrate execution to a different ISA is made probabilistically at run-time, thereby limiting an attacker's ability to chain gadgets reliably.

While randomization across heterogeneous-ISAs systematically removes the knowledge of what architecture the attacker is executing on, in the next section, we show how randomization within an ISA could further extend the effectiveness of our technique.

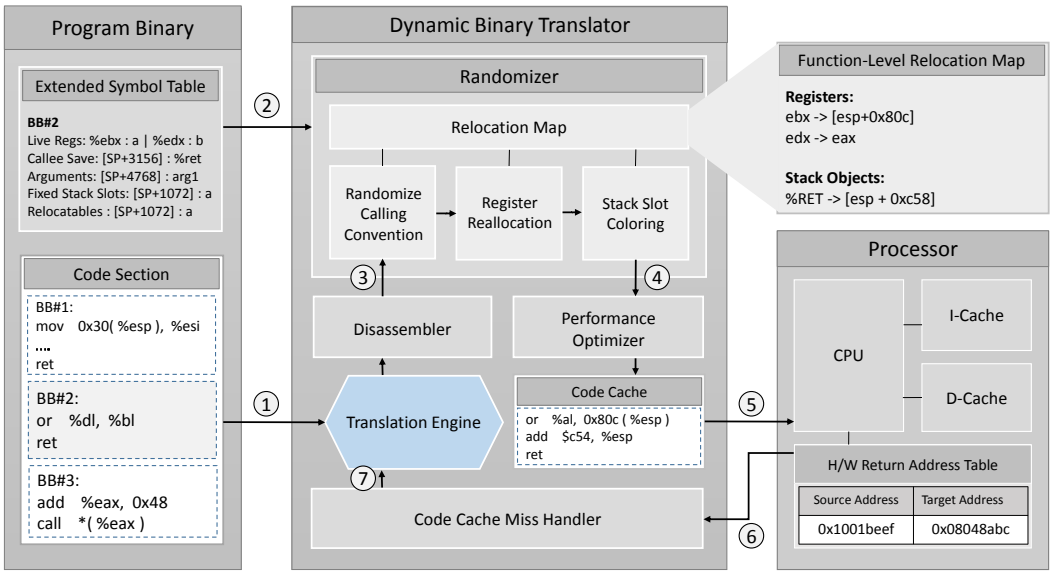

Figure 4.1. Program State Relocation Architecture

4.2 Program State Relocation

Program State Relocation (PSR) comprises a set of dynamic binary code transformations that can be easily deployed in any JIT-based system. The major goal of program state relocation is to shuffle program state (registers and memory) such that it is always found at the expected location during legitimate execution, but it is highly unlikely to be found by a ROP gadget that strays away from the legitimate control flow path.

As shown in Figure 4.1, the PSR runtime operates in a classic just-in-time dynamic translation mode, processing one basic block at a time. For each basic block in translation, it gathers information about the parent function, which is available from static analysis. Irrespective of the point of entry, the PSR runtime constructs a *relocation map* for every function, if it is being entered for the first time. The relocation map specifies the randomized calling conventions to be followed while calling the function, along with a set of randomized register allocation and stack slot coloring rules to be followed within that function.

As with classic DBT, translation is performed until an indirect or conditional jump is reached, at which point control is transferred to the translated code in the code cache. If a translation for the jump target is not available (a code cache miss), necessary transformations are applied as described above, and control is relinquished to the translated code. To ensure the code cache does not get compromised, we mandate that all return addresses stored on the stack point to original source code instead of the translated version. Furthermore, we make minor changes to the call and return instructions (macro-ops) to perform an extra cycle look-up in a hardware-maintained *Return Address Table* (RAT), in order to translate the source-level address to its corresponding translated version before making the actual control transfer.

The effect of program state relocation is that an object previously found in a register may be relocated to a different register or a random location on the stack, and vice-versa. Due to the sheer number of stack locations available to use for relocating an object, the number of possible dynamic code transformations (entropy) explodes, thereby rendering classic brute force attacks such as Blind-ROP [15] practically impossible on a system implementing PSR. Moreover, since the transformations happen at run-time rather than load-time, a PSR system will always re-randomize upon a crash or reboot, further strengthening its effectiveness.

4.3 Heterogeneous-ISA PSR

Instruction Set Randomization and Program State Relocation each represent strong defenses independently. However, we find that there is significant synergy between the two techniques, and one technique only amplifies the effectiveness of the other. Therefore, we combine them into one solid defense called “Heterogeneous-ISA Program State Relocation”. Figure 4.2 shows the high level architecture of Heterogeneous-ISA PSR.

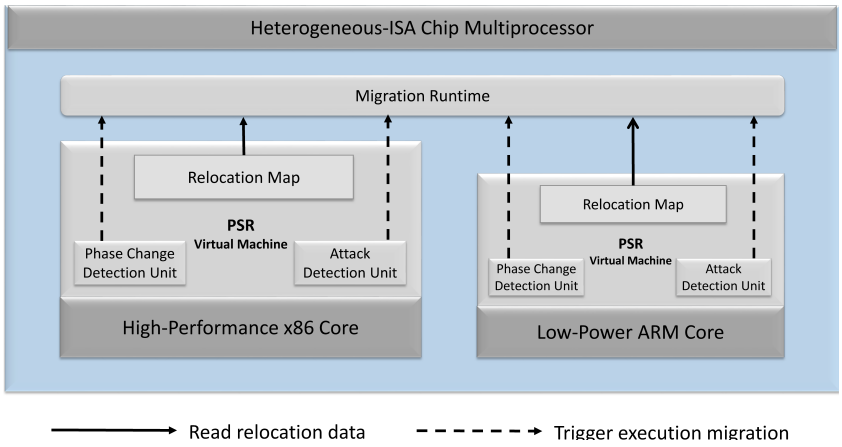

Figure 4.2. Heterogeneous-ISA Program State Relocation Architecture

The defense leverages a heterogeneous-ISA CMP composed of a low-power ARM core and a high-performance x86 core, that each run a virtual machine capable of performing program state relocation. To continue to reap the full performance/energy benefits of the heterogeneous-ISA CMP, we perform task migration only when an application phase change demands migration to a different ISA. Additionally, we perform non-deterministic execution migration between the two ISAs only when the PSR runtime detects a possible attempt to compromise security.

In our evaluation, we find that a code cache miss resulting from an indirect control transfer (including returns) is one of the key characteristics of a possible security breach. A code cache miss could result from one of two scenarios. In the legitimate execution scenario, the jump target is valid, but has not been translated yet (compulsory miss), or a translation for it was previously evicted from the code cache (capacity miss). In an attack scenario, the jump target points to a ROP gadget, and therefore a mapping does not exist in the PSR data structures. The PSR virtual machines make no effort to distinguish between the two scenarios. They instead migrate execution to a different ISA (with some probability) on every indirect control transfer that misses the code cache.

Like any JIT system with a sufficiently large code cache, one would expect code

cache misses to be infrequent once the application reaches a steady state in execution. Therefore, legitimate execution should experience no meaningful degradation in steady state performance. Furthermore, we perform multiple translations, one for each ISA, when an indirect control transfer results in a compulsory miss, further reducing miss events. This implies that, in steady state, an application will continue to execute on the ISA of its preference, because a translation for the jump target is found in either ISA.

In theory, an attacker could avoid migrating to a different ISA by using gadgets that are already translated indirect jump targets or function call sites, for which the PSR virtual machines already have a mapping in their internal data structures. In our evaluation, we find that the number of such gadgets is insufficient even for the simplest *execve* exploit.

Chapter 4, in part is currently being prepared for submission for publication of the material. Venkat, Ashish; Shamasunder, Sriskanda; Tullsen, Dean; Shacham, Hovav. The thesis author was one of the primary investigators and co-author of this material.

Chapter 5

Design and Implementation

In this chapter, we present the design and implementation details of Program State Relocation and discuss how our system behaves under different execution scenarios.

5.1 Program State Relocation

Program State Relocation is a set of transformations that relocate program state (registers and stack objects) within the same ISA. In our implementation, these transformations essentially randomize calling conventions, register allocation, and stack slot coloring. While most of these transformations can be accomplished by a mere change in the addressing mode, some transformations (e.g., procedure call/return) are slightly more involved and might require insertion of a small number of *move* instructions.

5.1.1 Addressing Mode Transformation

Each instruction in a basic block is modified to access its source and destination operands at their new locations, as specified by the function's relocation map. In most cases, this transformation is rather trivial and involves mere changing of addressing modes. If the ISA does not expose a certain addressing mode, the PSR virtual machine emulates it using additional instructions and register temporaries. For example, owing to the variety of addressing modes in x86, we use additional instructions only when more

than one operand of an instruction is relocated to memory.

5.1.2 Procedure Call Transformation

The PSR virtual machine instruments all procedure call and system call instructions to perform argument relocation and register spill/restore as specified by the callee's relocation map and the target ABI, respectively. As an optimization, the PSR virtual machine eliminates any redundant caller/callee register save and restore instructions. Furthermore, the virtual machine allocates 2 to 16 pages of randomization space on the stack in addition to the space already used by the callee's locals, temporaries, and spills, effectively providing 13 to 16 bits of entropy for every register or memory access. Note that return addresses are also relocated to random offsets, and therefore even a *nop* gadget that just performs a return incurs an entropy of at least 13 bits.

One of the biggest challenges with procedure call transformation is to preserve the live-ins and live-outs across function call sites, and correctly compute the caller/callee saves upon every function invocation. We take advantage of a single basic block look-ahead liveness analysis to accurately compute this information, and incorporate them into the randomized calling convention. A major source of ROP gadgets include the callee restore sequence that pops a bunch of callee save registers before returning back to the caller. To circumvent this, we perform a randomized scatter of callee saves (spray callee saves to random locations on the stack) at the function call site, and a randomized gather after return.

5.1.3 Indirect Control Transfer

Like any DBT system, the PSR virtual machine traps all indirect jumps into the translator. This implies there exist absolutely no indirect jumps translated into the code cache. As a software fault isolation measure, we terminate the process in case we find

an indirect jump target within the code cache's address range. Similarly, we disallow pointers to the code cache to exist as function pointers or return addresses on the stack. We handle function pointers in the same way as indirect jumps.

For function returns however, we always push the source return address on the stack, and take advantage of a hardware TLB-like structure called the return address table (RAT) that contains a mapping from source address (address of the function call site in the native binary) to target address (address of the function call site in the code cache). The call macro-op in the processor is modified to update the RAT with the right mapping, while the return macro-op is modified to perform return address translation as an extra step with a 1-cycle penalty. Upon a RAT miss, we conclude that there was a code cache miss and trap into the translator, for re-translation of that basic block.

5.2 PSR-aware Execution Migration

Our migration policy allows execution migration across heterogeneous ISAs in two specific scenarios. First, we migrate execution whenever an application's phase changes or the processor's current operating condition demands migration to another core. This is essential because it preserves the performance and energy advantages of a heterogeneous-ISA CMP. On the other hand, we also migrate execution, although probabilistically, when the PSR virtual machine suspects a security breach (specifically, when an indirect control transfer results in a code cache miss).

Prior work on heterogeneous-ISA execution migration suggests that we can be migration-safe at only 45% of the basic blocks [59]. To support instantaneous migration, they employ dynamic binary translation until a point of execution is reached, where the stack can be safely transformed. This implies that a ROP exploit that is composed entirely out of the remaining 55% of the basic blocks could completely bypass instruction set randomization.

To circumvent this, we re-purpose the original multi-ISA compilation infrastructure to support an on-demand execution migration. In essence, we transform only those objects on the stack that are absolutely necessary for executing instructions until the next control transfer (jump, call or return), and revert back to the original ISA to execute the next basic block. By doing so, we manage to be migration-safe for as much as 88% of the time. Furthermore, we completely avoid jumps to *unintentional* gadgets upon a code cache miss. We do this by taking advantage of an attack detection unit that disassembles from the last seen nearest address (or function boundary) to the program counter, up until the program counter itself. This is a minor change to the PSR virtual machine, which already does sophisticated liveness analysis.

Finally, we ensure that our migration strategy is PSR-aware, which means we not only transform an object from one ISA-form to another, but we fetch the object from its randomized location on one ISA and move it to its new randomized location on the other ISA.

5.3 Execution Scenarios

Legitimate execution. In a legitimate execution scenario, the procedure call transformation ensures that functions are always presented with relocated arguments. Furthermore, basic blocks are also presented with relocated live-ins since execution starts at the intended entry point of the function, thereby preserving the integrity of legitimate program execution.

5.3.1 Stack Unwinding

Libraries such as *libunwind* rely on compiler generated stack frame layout information to unwind the stack in exceptional scenarios such as *setjmp and longjmp*, and C++ exceptions. PSR seamlessly works with *setjmp and longjmp* due to the temporary

register spill/restore, performed as a part of the procedure call transformation.

However, C++ exceptions and other debugger routines unwind the stack frame-by-frame, inspecting stack objects at each frame, until the unwind target is reached. Performing PSR on such routines might lead to inconsistent program state. To prevent such inconsistencies, the PSR virtual machine instruments these unwind routines to use the same relocation map as the function that owns the frame being processed. This guarantees that frame objects are always accessed from their appropriate relocated addresses, irrespective of the control flow.

Furthermore, we force migration (and thus stack transformation) in the rare event when a *longjmp* is taken, but the corresponding *setjmp* was performed on a different ISA.

5.3.2 ROP attack

In the event of a ROP attack, the buffer overflow itself happens at a relocated stack address. Therefore, there is no guarantee that the return address is overwritten with the gadget address. In case the attacker manages to successfully overwrite the return address, she will find that the gadget at that address fails to work as intended. This is because the PSR virtual machine dynamically transforms every instruction in that gadget to access data from their randomized locations. Note that this is not just true for ROP attacks, but hold for jump-oriented programming, v-table hijack, and other variants. PSR inherently defeats return-into-libc because of the randomized calling conventions.

5.3.3 Crash/Reboot scenarios

To guarantee high quality of service and robustness, most servers re-spawn worker threads upon a crash or a reboot. Several brute force attacks such as Blind-ROP exploit this property of servers to mount repeated attacks until they become compromised. These attacks typically bank on using information leaked in a previous attempt, in order to

reduce the overall time-to-attack. This is possible because a process randomized at load-time typically does not get re-randomized every time it spawns a thread. However, a PSR virtual machine performs randomization at run-time, which means we have the ability to re-randomize upon re-spawn. Note that this extends to the PSR virtual machines on both ISAs. Therefore, each time a worker thread re-spawns, the attacker is presented with a re-randomized version of the code cache on both ISAs.

Chapter 5, in part is currently being prepared for submission for publication of the material. Venkat, Ashish; Shamasunder, Sriskanda; Tullsen, Dean; Shacham, Hovav. The thesis author was one of the primary investigators and co-author of this material.

Chapter 6

Threat Model

To evaluate the effectiveness of our defense, we make several conservative assumptions about our threat model.

6.1 Complete Disclosure

We assume that the attacker has full knowledge of the inner workings of our defense mechanisms. We also assume that the attacker has unfettered access to the binary, source code, and complete control flow graph of the program in execution. Consequently, the attacker has a complete list of all potential ROP/JOP gadgets in the binary, and is capable of mounting attacks ranging from classic ROP [51] to just-in-time code reuse (JIT-ROP) [54] attacks.

6.2 Just-in-time Code Reuse

We assume that the attacker has the ability to snoop into a program's memory in order to bypass address space and fine-grained code randomization. To this end, we assume the program in execution exhibits one or more vulnerabilities that allow an attacker to (a) write to memory (by means of a stack/heap based overflow), and (b) read an arbitrary number of bytes from any memory location, using a single leaked memory

disclosure.

6.3 Brute Force Attacks

We also assume that the system is susceptible to brute force attacks such as Blind-ROP [15]. To this end, we model a system as described by Shacham, et al. [52] that assumes a program executing as a child thread, whose parent re-spawns it upon on a crash. We do not assume any defense mechanism that monitors the frequency of such an event to detect ROP attacks. We instead use it as a metric to demonstrate the effectiveness of PSR against brute force.

6.4 JIT-Spraying

JIT-spraying techniques exploit the just-in-time compilation functionality to generate predictable chunks of exploit code in the text section, using carefully crafted JavaScript or ActionScript called GaJITS [49]. We find that PSR implicitly defeats such attacks because GaJITS undergo significant mangling when subjected to program state relocation.

Although we assume non-executable memory, we take no extra steps to prevent JIT Spraying attacks [16] that exploit the nature of JIT compilation to inject executable code as data. We assume that the program is susceptible to such attacks, but find that PSR implicitly defeats them due to its nature of run-time randomization, rather than at load time.

Chapter 6, in part is currently being prepared for submission for publication of the material. Venkat, Ashish; Shamasunder, Sriskanda; Tullsen, Dean; Shacham, Hovav. The thesis author was one of the primary investigators and co-author of this material.

Chapter 7

Evaluating Effectiveness

Heterogeneous-ISA Program State Relocation, at the heart of it, is a randomization based defense. A key metric of randomization based defenses in entropy [52], or, the amount of randomness it introduces into the system. Therefore it is quintessential to evaluate PSR for the entropy it provides. Entropy, though, is a theoretical estimate and only represents the potential of a defense technique in resisting attacks. A true test of any defense mechanism is its ability to thwart real attacks, ranging from simple ones like classic ROP to more sophisticated ones like JIT-ROP.

The biggest challenge in evaluating a complex system like PSR is that the sheer number of randomizable states that it offers can often make even simple experiments practically infeasible. For example, the average number of randomizable states for each register/stack location under PSR is 2^{13}. Evaluating it against only a handful of popular exploits is more practical but is hardly a guarantee of its effectiveness. What then we need is a metric that models the vulnerability of a system to attacks realistically, and measure how PSR can make the system less vulnerable.

An important characteristic of any attack is that it requires the victim program to expose a reasonable *attack surface* to exploit. In the context of ROP, the attack surface is represented by the number of gadgets available in a program that facilitate the construction of a successful exploit. The goal of every randomization defense is to

Table 7.1. Attack Surface: Symbols and Definitions

Symbol	Definition
G_{ROP}	Size of the attack surface for a classic ROP attack.
G_{mod}	Number of gadgets modified by PSR.
G_{new}	Number of gadgets introduced by PSR.
$G_{JIT-ROP}$	Size of the attack surface for a JIT-ROP attack.
$G_{JIT-ISA}$	Size of the attack surface for a JIT-ROP attack in Heterogeneous-ISA PSR.

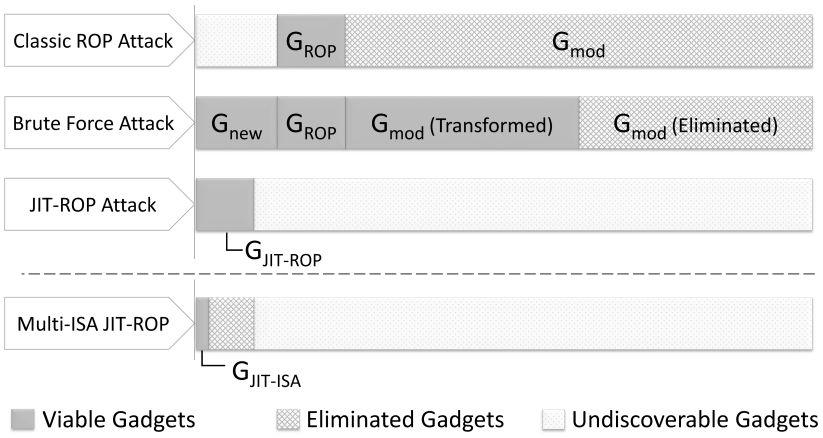

Figure 7.1. Attack Surface of a Victim Program

reduce the attack surface (both in terms of availability and functionality), in order to limit the attacker's ability to construct meaningful exploits. Therefore, we use the attack surface of victim programs as a measure of its vulnerability, and measure how PSR can reduce this attack surface. Note that the sample space of possible exploits for a given attack surface can be considerably smaller or even non-existent, therefore we believe the attack surface is a conservative estimate of a program's vulnerability to ROP.

Figure 7.1 represents a victim program's attack surface for different types of attacks while running on our architecture. Table 7.1 introduces a list of symbols and their definitions that we use to represent key elements of an attack surface through the rest of this section.

Since every exploit requires some program state in the form of either registers or

stack objects, we designate any gadget that successfully populates a register with attacker intended value as *viable*. We evaluate every gadget for its viability on a system, without and with PSR, to measure the attack surface for three major classes of attacks: (a) classic ROP, (b) brute-force, and (c) JIT-ROP.

7.1 Classic ROP Attack

Classic ROP attacks involve using the set of gadgets discovered from analysing the binary to construct an exploit. Under this form of attack the attacker is either unaware of any underlying randomization that has been performed or is unable to bypass it. Under PSR we expect a majority of the initially discovered viable set of gadgets to be transformed. G_{mod} represents these gadgets that have been modified by PSR and hence rendered useless for classic ROP. Since PSR is a randomization scheme, there is always a chance that some gadgets remain in their original form. G_{ROP} represents this set of gadgets that remain untouched and viable for attackers.

7.2 Brute Force Attacks

As illustrated in Figure 7.1, PSR modifies a majority of the gadgets that were previously available for ROP. These gadgets (G_{mod}), by virtue of PSR's transformations, have either been modified in a way that they no longer perform the attacker intended action, or have been completely eliminated. The former of these is a *viable* candidate for a brute force attack since it performs useful computation, just not what an attacker expects it to. Also *viable* for a brute force attack is any gadget introduced by the randomization itself, denoted by G_{new} in our discussion. This is true of almost every fine-grained randomization technique that breaks gadgets by relocating them or transforming them. The attack surface for brute force comprises every gadget available in the program, since there is no way to ascertain which ones will transform to be *viable* gadgets. Note

that the set of *viable* gadgets for brute force includes G_{ROP}, G_{mod} (transformed gadgets only), and G_{new}. We can expect that a sizeable portion of all gadgets are *viable* for brute-force, and therefore require thorough evaluation. Although some existing ROP defenses dismiss brute force as impossible assuming that an operating system would detect multiple crashes, or that the user would not re-run a crashing application, it has been proven that brute force remains a viable option if the application is vulnerable to repeated attacks [15, 52].

Algorithm 1. Brute Force Simulation

$G = \{g_1, g_2 \ldots g_n\}$ /* Set of n available gadgets. */
$R = \{r_1, r_2 \ldots r_m\}$ /* Set of m registers to load. */
$P = \emptyset$ /* Set of successfully populated registers. */
$X = ()$ /* List of chosen gadgets for the attack. */
$Y = ()$ /* List of return address locations for chosen gadgets. */
$A(g)$ is the randomized return address for gadget g

for all $i = 1$ to m **do**
    r_i is the register to populate
    find g_j in G s.t. g_j populates register r_i,
        does not clobber any register s in P, and
        $A(g_j) = \min_{k=1\ldots n} A(g_k)$
    $P = P + \{r_i\}$
    $X = X + \{\mathrm{j}\}$
    $Y = Y + \{A(g_j)\}$
end for

Let B be the number of attempts to populate all registers, then
    for an average frame size of f
$B = Y[0] + f.X[0] + nf.Y[1] + nf^2.X[1] + \ldots + n^3 f^4.X[3]$

To evaluate the system against brute force attacks while keeping the experiment tractable, we analyze each gadget to gather data about every perturbation it produces on the state of the program, at a randomly chosen point in its execution. We then simulate a brute force attack by running this data through Algorithm 1. Cheng, et al. [20] showed

that the shortest aligned gadget chain generated by gadget compilers such as Q [50] is 17, but to establish the effectiveness of PSR, we consider a much smaller four-gadget shellcode exploit that performs the system call *execve*(), which in theory should be easier to brute force by several orders of magnitude. Although the run-time nature of PSR transformations involve re-randomization upon crash, to keep the experiment tractable, we make the conservative assumption that a failed attempt does not result in re-randomization, and thereby tip the scales in the attacker's favor.

Algorithm 1 simulates a brute force attack to populate the four registers (*eax*, *ebx*, *ecx*, and *edx*) necessary to perform the *execve*() system call with attacker provided values on the stack. On a system protected by PSR, all program state (registers and stack objects, including the return address) is relocated to a random register or a stack location. Therefore, such an attack should brute force three independent variables in the system: (a) the gadget to execute, (b) relative position(s) of data on the stack, as required by the gadget, and (c) relative position of the return address on the stack, required to chain the next gadget. The attacker should brute force the gadget itself, because it is difficult to determine the potential *viability* of a gadget that will inevitably be subject to PSR. Therefore, we brute force every gadget discovered by the Galileo algorithm. The data for each gadget (the value to load into a register) and the return address, both share the same stack frame. In an unsecured system their locations can be easily determined, but with PSR, they can lie anywhere within a stack frame.

To maximize the success of a gadget, we *spray* the data for the gadget on the entire stack frame and brute force the location of the return address within the frame. We model our attack to populate one register at a time, in order to *spray* an entire stack frame with the data for one register, thereby increasing its chances of being read by a gadget. Since we assume the attacker has insight into the inner workings of PSR, we assume a frame size of 8KB, at which PSR provides substantial security benefits at an

acceptable degradation in performance. In our algorithm, we also account for register and stack clobbering to ensure that a gadget does not destroy previously established state. The algorithm stops searching for more *viable* gadgets as soon it finds a four-gadget shellcode exploit.

It is worth noting that our method of simulating brute force loosely resembles the Blind-ROP algorithm [15] that finds *viable* gadgets when an attacker has no knowledge of the binary or source code. The key difference is that Blind-ROP relies on the target binary respecting traditional calling conventions and stack layout, whereas in a PSR-protected system, we can make no such assumptions. Therefore, a Blind-ROP attack on a system secured with PSR essentially translates to a version of our brute force attack and will require a similar number of attempts to succeed.

7.3 Just-In-Time Code Reuse

In most randomization techniques, the program code is rewritten at load-time to eliminate or modify gadgets. They attempt to reduce the attack surface by hiding the randomized version of the code from the attacker. But in the presence of a memory disclosure vulnerability, the entire memory image of a process is now accessible to the attacker, including the randomized code, thereby providing JIT-ROP with an attack surface similar in size to that of classic ROP.

Program State Relocation on the other hand randomizes code at run-time, rather than at load-time, using dynamic binary translation. Owing to the just-in-time nature of PSR, only the steady state program code that has already been randomized by PSR, and is present in the code cache, remains vulnerable to JIT-ROP. To quantify this vulnerability we measure the attack surface for a JIT-ROP attack, $G_{JIT-ROP}$, by identifying the set of viable gadgets that lie within the code cache. It is important to note that this only represents the attack surface on a single-ISA machine.

In the proposed heteroegeneous-ISA architecture, the PSR virtual machine migrates execution across ISAs for every indirect control transfer (including returns) that misses the code cache. Therefore, to measure the true attack surface for a JIT-ROP attack under this architecture we only need to account for those gadgets that are viable, present in the code cache, and are preceded by a call instruction, therefore representing a valid return target.

Chapter 9, in part is currently being prepared for submission for publication of the material. Venkat, Ashish; Shamasunder, Sriskanda; Tullsen, Dean; Shacham, Hovav. The thesis author was one of the primary investigators and co-author of this material.

Chapter 8

Methodology

Evaluating PSR against the metrics defined in chapter 7 requires the use of several tools and techniques, including some that we developed for this sole purpose. In this section we describe the experimental methodology used, along with a description of the tools used to facilitate it.

8.1 Gadget Discovery and Entropy

We use the 'Galileo' algorithm described by Shacham, et al. [51] to mine each benchmark for every possible instruction sequence that ends with a return instruction. The algorithm searches for gadgets in a program's binary by finding return instructions (*ret* in x86) and working its way backwards byte by byte to build a trie of all possible instructions leading up to it. This way we not only capture all instruction sequences in the source binary ending with a return but also unintentional instruction sequences that are an artifact of x86's unaligned instruction access and variable length instructions.

Measuring entropy turns out to be more challenging since there is no universal definition of entropy for randomization techniques. Entropy is often described as the amount of randomness a mechanism introduces into the system. For example, the entropy for Address Space Layout Randomization (ASLR) [57] is defined as the number of possible locations to which each gadget can be relocated by randomizing the address of

each program section in its address space. On a 32-bit machine, ASLR can relocate a section to 2^{32} possible locations, therefore it is said to provide 32 bits of entropy.

Since PSR randomizes program state, the location of registers and stack slots, we define the entropy for a gadget under PSR as the number of possible states that all the program variables in the gadget can occupy. That is, the sum total of all possible locations that each register and stack slot can be relocated to. The entropy of a gadget is calculated as:

$$E_g = (Num\_Regs_g + 1) * log_2 Frame\_Size$$

where Num_Regs_g is the number of unique registers accessed by the gadget (including stack pointer for instructions that operate on the stack), and $Frame\_Size$ is the frame size of the function where the gadget resides. Num_Regs_g is incremented by 1 to account for the randomized return address location on the stack, since every gadget ends in a *ret*.

Since this number depends on the number of registers and stack slots used by each gadget and since each program contains different types of gadgets, the entropy varies based on the program. We calculate the entropy of a program as the average of the entropies of all its gadgets.

8.2 Simulating Brute Force

As described in chapter 7, in order to make experiments such as brute force tractable we analyse the gadgets individually, offline. To replicate the conditions of a program in execution we use a snapshot (or checkpoint) of each benchmark after executing 1 billion instructions. We skip the first billion instructions in order to avoid any initialization code and correctly model steady state behavior.

We use a modified version of the PSR binary translator that restores program state from this checkpoint, and analyses each gadget. Before executing a gadget we perform certain steps to recreate an attack environment. In order to simulate an intelligent attacker we pre-populate the entire stack frame (8KB in our tests) with a magic word that we intend to populate the registers with. We also record the state of each register and the current stack frame along with two adjacent stack frames. The DBT then executes the gadget until either completion (return) or failure, and in either case, traps back to the binary translator. In some cases a gadget may cause a catastrophic failure and crash the translator. We treat these like any other failed gadgets. After execution, we once again record the state of all gadgets and the few stack frames under observation. This allows us to not only identify which registers a gadget successfully populates, but also ones that it clobbers. We then check for any differences to identify the changes to program state from a gadget's execution. We also note the randomized return address and the randomized stack frame size for each gadget.

We analyse each gadget in a benchmark individually and build a database of all perturbations caused by them. We then run algorithm 1 defined in chapter 7 over this raw data using a perl script that calculates the number of attempts required to successfully populate all 4 registers required for a shellcode exploit, taking the following factors into account: 1) The position of the gadget in our list of gadgets, 2) Registers populated with the value from the stack, 3) Registers clobbered, 4) Stack slots clobbered, and 5) Location of return address on the stack frame.

8.3 Simulating JIT-ROP

As described in chapter 7, JIT-ROP attacks under PSR are limited to the gadgets available in the code cache of the binary translator. In order to identify the set of viable gadgets in the code cache, we first need to identify the gadgets in the code cache.

Once again we use a gem5 checkpoint of of each benchmark after executing 1 billion instructions. We analyse the checkpoint to identify all the instructions that have been placed in the code cache by the translator at this point in time. We then compare this list to the list of viable gadgets we constructed by analysing each gadget as described in the previous section. We note this subset of gadgets as the set of gadgets that are useful for a JIT-ROP attack. Further we use an object dump of the binary to identify all call preceded instructions and compare this with the previous list to generate a list of all gadgets that are successful under heterogeneous-ISA PSR.

8.4 Correctness

Software verification is a major challenge for any code randomization defense, and more so for heterogeneous-ISA PSR because it not only randomizes a program's architectural state, but randomizes the architecture itself. To ensure we preserve the semantics of a program at all times, we perform two types of sanity checks.

First, we periodically examine the register and stack contents of a randomized program in execution, and compare it against the unrandomized version, accounting for relocation of variables and stack locations. Our test infrastructure has the ability to tune the frequency of this sanity check at the function, basic block, and instruction levels. Second, we ensure that the migration runtime has appropriately transformed the program's architectural state by comparing it against a *checkpoint* of the same program that has been executing on the migrated-to ISA from the time of its instantiation. We do so by leveraging the gem5 [14] architectural simulator to generate checkpoints that capture the dynamic execution state of a program for each ISA. Through these checks, we have verified that the program state is as expected in each of the checkpoints we use for our experiments, for all of the benchmarks.

8.5 Experimental setup

Table 8.1. Benchmarks along with description

Benchmark	Description
bzip2	Compression
gobmk	Artificial Intelligence: go
hmmer	Search Gene Sequence
lbm	Fluid Dynamics
libquantum	Physics: Quantum Computing
mcf	Combinatorial Optimization
milc	Physics: Quantum Chromodynamics
sphinx3	Speech recognition

We use the SPEC CPU2006 integer and floating-point C benchmarks to evaluate the proposed defense. We exclude *gcc* and *sjeng* from this set because they perform dynamic memory allocation on the stack either using the *alloca* library function, or by passing variable-length array parameters. While our multi-ISA compilation and runtime infrastructure is capable of working with variable-size stack frames, our current PSR implementation does not support this feature yet. All benchmarks are compiled using an LLVM-based multi-ISA compiler at the -O3 optimization level. To model a heterogeneous-ISA CMP, we use the gem5 [14] architectural simulator. The processor model of the ARM core is based on the low-power Cortex A-15, while the x86 core is modeled after the high performance Core-i7. Table 8.1 describes the set of benchmarks used in our experiments.

Chapter 9

Results

This chapter discusses the results of our experiments to evaluate the effectiveness of PSR as a defense technique and our inferences from the same.

9.1 Classic ROP Attacks

Figure 9.1 shows the extent to which PSR reduces the attack surface for classic ROP-style attacks, including return-into-libc, jump-oriented programming, and v-table hijack. We observe that the sheer amount of randomization that each gadget undergoes guarantees that only an insignificant portion of the attack surface remains unaltered. In particular, PSR reduces the attack surface of classic ROP (G_{ROP}) by an average of 98.51%. We note that although the remaining 1.49% is unmodified by PSR, the attacker has no way of determining which gadgets they are, thereby rendering classic ROP attacks infeasible.

9.2 Brute Force Attacks

Table 9.1 shows the results of our brute force simulation. We observe that PSR successfully renders brute force attacks computationally infeasible, by a considerably large margin. We find that, on an average, a gadget has between six and seven ran-

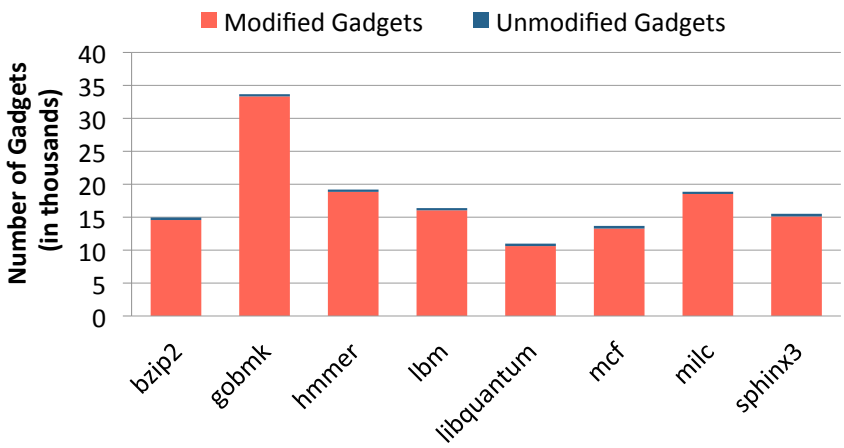

Figure 9.1. Classic ROP Attack Surface Reduction

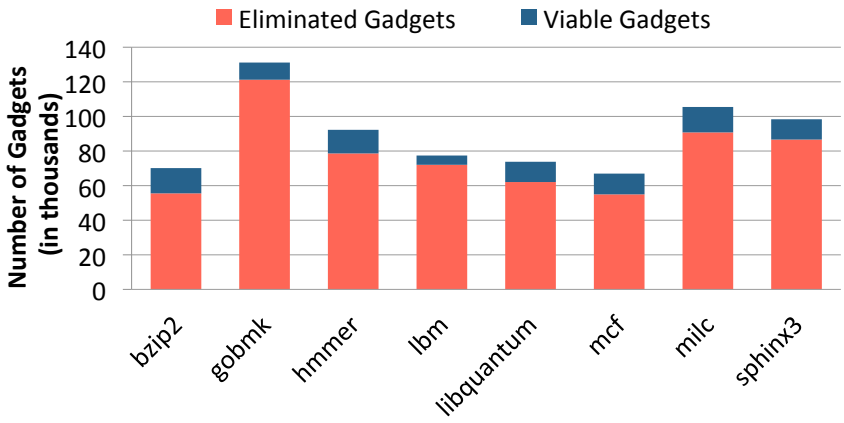

Figure 9.2. Brute Force Attack Surface Reduction

domizable parameters which could potentially include registers, stack objects, and at least one address on the stack to place the (return) address of the next gadget. In our configuration of 8KB sized stack frames, each parameter has 2^{13} randomizable states, resulting in an average entropy of 87 bits per gadget. Even if a vulnerability allowed an indefinite number of attempts, with each attempt only taking a nanosecond, we find that it is computationally infeasible to perform such a brute force attack with state-of-the-art computing infrastructure. In fact, such an attack would remain computationally infeasible on future processors targeted at exascale computing.

Table 9.1. Inferences from Brute Force Simulation

Benchmark	Randomizable Params (avg)	Entropy	Number of Attempts
bzip2	6.76	88	2.34 x 10^{33}
gobmk	6.53	85	2.87 x 10^{34}
hmmer	6.69	87	1.16 x 10^{34}
lbm	6.92	90	3.90 x 10^{34}
libquantum	6.76	88	6.45 x 10^{33}
mcf	6.69	87	1.71 x 10^{34}
milc	6.46	84	2.92 x 10^{34}
sphinx3	6.92	90	8.68 x 10^{33}

9.3 Just-In-Time Code Reuse Attacks

We find that $G_{JIT-ROP}$, the number of gadgets already randomized by PSR, accounts for only 1.18% of all classic ROP gadgets and 1.64% of those *viable* for brute force, thereby severely constraining the attack surface. Figure 7.1 shows this reduction in attack surface for JIT-ROP. Note that a majority of gadgets are now *undiscoverable*, since they lie outside the code cache.

Although the attack surface has been considerably reduced, the gadgets that comprise $G_{JIT-ROP}$ could potentially be enough to mount a JIT-ROP attack, although a weak one. Recall from Chapter 3 that the PSR virtual machines suspect a security violation when an indirect control transfer (including returns) misses the code cache, and subsequently migrate execution to a different ISA, albeit probabilistically. Note that the PSR virtual machine can find in its internal structures only those indirect jump targets and function call sites that have been translated so far, and will result in a code cache miss for all others. Any *viable* gadget for JIT-ROP must avoid migration to a different ISA, and therefore begin at an already translated indirect jump or function call site. This imposes serious limitations on the JIT-ROP attack surface that has already been weakened by PSR. Figure 7.1 represents this as $G_{JIT-ISA}$, the true size of the attack surface for a JIT-ROP attack on heterogeneous-ISA PSR.

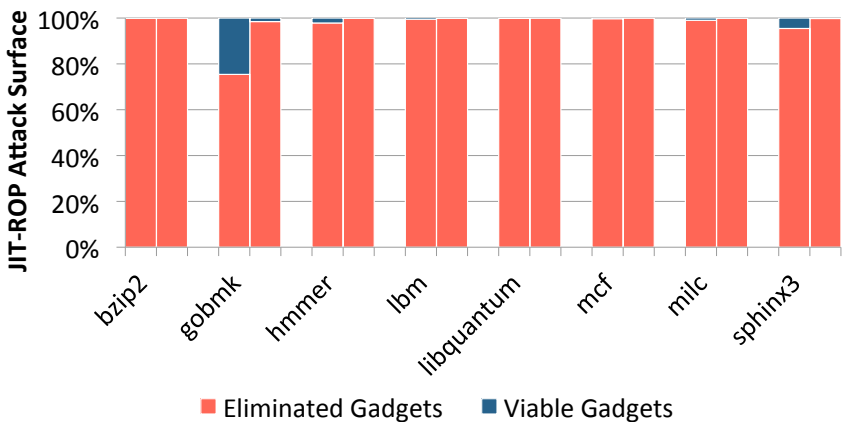

Figure 9.3. JIT-ROP Attack Surface Reduction on (a) Single-ISA PSR, and (b) Heterogeneous-ISA PSR

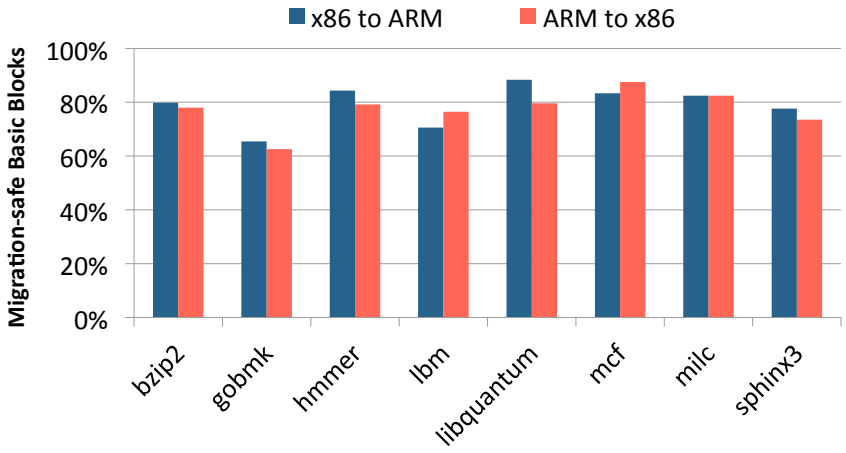

Figure 9.4. Percentage of Migration-Safe Basic Blocks

Figure 9.3 shows the reduction in attack surface for each benchmark under both single-ISA and heterogeneous-ISA PSR. We find that only 8.87% of $G_{JIT-ROP}$, or 0.09% of all ROP gadgets qualify as *viable* for $G_{JIT-ISA}$. Furthermore, as shown in Figure 9.4, we note that our infrastructure is capable of being migration-safe on an average of 78% of the time, in either direction. This implies that gadgets in the remaining 22% of the basic blocks are still *viable* candidates for JIT-ROP. However, we find that these remaining gadgets are insufficient to even construct a four-gadget shellcode exploit, let alone complex exploits.

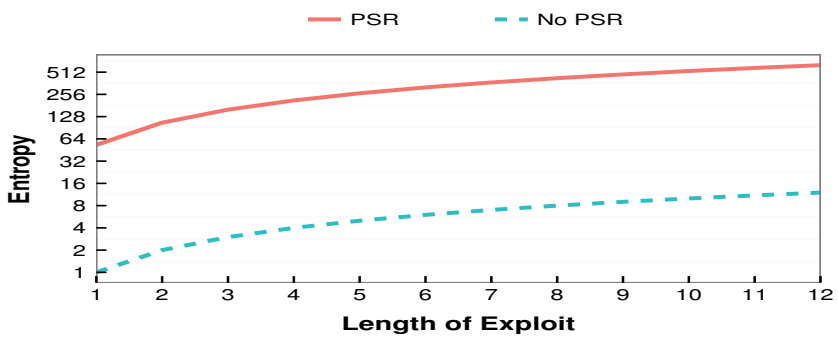

Figure 9.5. Brute Forcing JIT-ROP on a Heterogeneous-ISA CMP

To further explore the synergy between program state relocation and instruction set randomization, we analyze a brute force JIT-ROP attack on two distinct systems that both implement heterogeneous-ISA execution migration. On the first system, we randomly choose to migrate between heterogeneous ISAs upon execution of every return or indirect jump instruction (irrespective of a code cache miss). On the second, we enforce PSR along with heterogeneous-ISA migration, in which case we randomly migrate between heterogeneous ISAs only when an indirect control transfer misses the code cache.

Figure 9.5 shows the results of this experiment. We make two important observations. First, brute forcing both systems becomes exponentially harder as the length of the gadget chain increases, restricting the attacker's ability to construct complex exploits. Second, the just-in-time nature of PSR inherently enables re-randomization upon a crash. Therefore, the attacker is always presented with a re-randomized version of the code cache on both ISAs, for every brute force attempt. This significantly boosts the entropy of the system that implements heterogeneous-ISA program state relocation.

9.4 Heterogeneous-ISA attacks.

By removing the assumption of the underlying ISA, heterogeneous-ISA execution forces an attacker to explore attacks that are either architecture independent, or can predict which architecture the program is currently running on. Cha, et al. [18] show that it is possible to compile Platform Independent Programs (PIP) that can run on multiple ISAs. PIP uses 'gadgets' that encode a 'header' and separate instruction sequences for each ISA. The header contains an architecture agnostic bitstring that diverts control to instructions of the right ISA. Traditionally, PIP requires code injection or JIT-spraying. Code injection is prevented by the $W \oplus X$ protection implemented in most modern processors. We implicitly defeat JIT-spraying because all JIT-compiled code including JIT-sprayed gadgets undergo program state relocation.

An attacker who is aware of heterogeneous-ISA PSR could hypothetically construct more tailored attacks that interleave gadgets from both ISAs. For example, one could craft an exploit that alternates gadgets between x86 and ARM, such that the ARM gadgets are all *nop*s that end with an indirect jump to force execution back to x86 while not clobbering already established state. While such an attack does not guarantee success, it could improve an attacker's chances. We have been unable to identify any systematic way to construct such exploits with a practical chance of success.

Chapter 9, in part is currently being prepared for submission for publication of the material. Venkat, Ashish; Shamasunder, Sriskanda; Tullsen, Dean; Shacham, Hovav. The thesis author was one of the primary investigators and co-author of this material.

Chapter 10

Conclusion

As computers become ubiquitous in every aspect of our lives, computer security becomes equally important. Attacks on computers have gotten progressively evasive and malicious, and one of the most prominent among them is return-oriented programming (ROP). While many defense mechanisms have been proposed and some even commercially distributed, none can comprehensively address the threat that ROP and its several variants pose.

In this work we have leveraged the power of heterogeneous-ISA computing and a novel randomization technique named program state relocation to propose a new architecture dubbed Heterogeneous-ISA Program State Relocation that shows promise in mitigating ROP to a large extent. Through our experiments and analysis we have demonstrated that this architecture can serve as an effective defense against ROP, brute forced ROP, and Just-In-Time ROP. Heterogeneous-ISA attacks are a new breed of attacks that could potentially threaten this architecture but the possibility and practicality of mounting such an attack is yet to be explored.

We have shown how metrics such as attack surface area and entropy can be used to quantify the vulnerability of a system and the strength of a defense technique, respectively. We have also presented a technique to approximate the effects of an attack on a system, and use this information to perform exhaustive experimental evaluation of

randomization based defense mechanisms in a tangible manner.

In conclusion, we want to draw attention to the fact that architectural defenses can be very powerful, if harnessed effectively. Through Heterogeneous-ISA Program State Relocation, we have shown that by decoupling the architectural state with the micro-architectural state, we can extract significant security benefits that are unique to this abstraction of computing.

Bibliography

[1] HSA: A New Architecture for Heterogeneous Computing. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/HSA_TIRIAS_Whitepaper_Final_1-28-14.pdf.

[2] Ropper. https://scoding.de/ropper/.

[3] Ropshell. http://ropshell.com.

[4] 2nd Generation Intel Core vPro Processor Family. Technical report, Intel, 2008.

[5] The future is fusion: The Industry-Changing Impact of Accelerated Computing. Technical report, AMD, 2008.

[6] The Benefits of Multiple CPU Cores in Mobile Devices. Technical report, NVidia, 2010.

[7] Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance. Technical report, NVidia, 2011.

[8] M Abadi, M Budiu, U Erlingsson, and J Ligatti. Control-flow integrity. *Proceedings of the 12th ACM . . .* , 2005.

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

[10] S Andersen and V Abella. Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. 2004.

[11] A Baratloo, N Singh, and TK Tsai. Transparent Run-Time Defense Against Stack-Smashing Attacks. *USENIX Annual Technical . . .* , 2000.

[12] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015.

[13] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE computer*, 2007.

[14] N Binkert, B Beckmann, and G Black. The gem5 simulator. *ACM SIGARCH . . .* , 2011.

[15] A Bittau and A Belay. Hacking blind. *Security and Privacy . . .* , 2014.

[16] D Blazakis. Interpreter exploitation: Pointer inference and JIT spraying. *BlackHat DC*, 2010.

[17] D Bouthaina, M Baklouti, S Niar, and M AID. Shared hardware accelerator architectures for heterogeneous MPSoCs. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*, July 2013.

[18] SK Cha, B Pak, D Brumley, and RJ Lipton. Platform-independent programs. *. . . of the 17th ACM conference on . . .* , 2010.

[19] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10*, page 559, New York, New York, USA, October 2010. ACM Press.

[20] Y Cheng, Z Zhou, and M Yu. ROPecker: A generic and practical approach for defending against ROP attacks. *. . . on Network and . . .* , 2014.

[21] C Cowan, C Pu, D Maier, and J Walpole. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *Usenix Security*, 1998.

[22] K Van Craeynest and A Jaleel. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). *ACM SIGARCH . . .* , 2012.

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

[24] L Davi and C Liebchen. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. *. . . Systems Security . . .* , 2015.

[25] M DeVuyst, A Venkat, and DM Tullsen. Execution migration in a heterogeneous-ISA chip multiprocessor. *ACM SIGARCH Computer . . .* , 2012.

[26] I Fratrić. ROPGuard: Runtime prevention of return-oriented programming attacks. 2012.

[27] P Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 2011.

[28] MD Hill and MR Marty. Amdahl's law in the multicore era. *Computer*, 2008.

[29] J Hiser, A Nguyen-Tuong, and M Co. ILR: Where'd My Gadgets Go? *Security and Privacy . . .* , 2012.

[30] M Kayaalp and M Ozsoy. Branch regulation: Low-overhead protection from code reuse attacks. *. . . (ISCA), 2012 39th . . .* , 2012.

[31] T Kornau. Return oriented programming for the ARM architecture. *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[32] R Kumar and KI Farkas. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. *. . . , 2003. MICRO-36. . . .* , 2003.

[33] R Kumar and DM Tullsen. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. *. . . Computer Architecture . . .* , 2004.

[34] Rakesh Kumar, Dean M Tullsen, N P Jouppi, and P Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11), 2005.

[35] T Li, P Brett, and R Knauerhase. Operating system support for overlapping-ISA heterogeneous multi-core architectures. *. . . (HPCA), 2010 IEEE . . .* , 2010.

[36] D Lo, L Cheng, and R Govindaraju. Towards energy proportionality for large-scale latency-critical workloads. *Proceeding of the 41st . . .* , 2014.

[37] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd International Symposium on*. IEEE, 2015.

[38] TR Maeurer and D Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and . . .* , 2005.

[39] JP McGregor and DK Karig. A processor architecture defense against buffer overflow attacks. *. . . Research and Education . . .* , 2003.

[40] V Mohan, P Larsen, and S Brunthaler. Opaque control-flow integrity. *Symposium on Network . . .* , 2015.

[41] K Onarlioglu, L Bilge, and A Lanzi. G-Free: defeating return-oriented programming through gadget-less binaries. *Proceedings of the 26th . . .* , 2010.

[42] A One. Smashing the stack for fun and profit. *Phrack magazine*, 1996.

[43] H Ozdoganoglu. SmashGuard: A hardware solution to prevent security attacks on the function return address. *Computers, IEEE . . .* , 2006.

[44] A Ozment and SE Schechter. Milk or wine: does software security improve with age? *Usenix Security*, 2006.

[45] V Pappas. kBouncer: Efficient and transparent ROP mitigation. 2012.

[46] V Pappas. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. *Security and Privacy (SP . . .* , 2012.

[47] A Putnam and AM Caulfield. A reconfigurable fabric for accelerating large-scale datacenter services. *. . . Architecture (ISCA), . . .* , 2014.

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

[49] C Rohlf and Y Ivnitskiy. Attacking clientside jit compilers. *Black Hat USA*, 2011.

[50] EJ Schwartz, T Avgerinos, and D Brumley. Q: Exploit Hardening Made Easy. *USENIX Security . . .* , 2011.

[51] Hovav Shacham. The geometry of innocent flesh on the bone. In *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*, page 552, New York, New York, USA, October 2007. ACM Press.

[52] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security - CCS '04*, page 298, New York, New York, USA, October 2004. ACM Press.

[53] E Shioji, Y Kawakoya, M Iwamura, and T Hariu. Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks. *Proceedings of the 28th . . .* , 2012.

[54] KZ Snow and F Monrose. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. *Security and Privacy . . .* , 2013.

[55] L. Szekeres, S. McCamant, and Dawn Song. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, May 2013.

[56] PaX Team. PaX. http://pax.grsecurity.net.

[57] PaX Team. PaX address space layout randomization (ASLR). 2003.

[58] G Varsamopoulos. Trends and effects of energy proportionality on server provisioning in data centers. ... *Computing (HiPC), 2010* ..., 2010.

[59] A Venkat and DM Tullsen. Harnessing ISA diversity: design of a heterogeneous-ISA chip multiprocessor. *Computer Architecture (ISCA), 2014* ..., 2014.

[60] R Wartell, V Mohan, KW Hamlen, and Z Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. ... *of the 2012 ACM conference on* ..., 2012.