HIPStR – Heterogeneous-ISA Program State Relocation

Ashish Venkat   Sriskanda Shamasunder   Hovav Shacham   Dean M. Tullsen

University of California, San Diego

{asvenkat | sshamasu | hovav | tullsen}@cs.ucsd.edu

Abstract

Heterogeneous Chip Multiprocessors have been shown to provide significant performance and energy efficiency gains over homogeneous designs. Recent research has expanded the dimensions of heterogeneity to include diverse Instruction Set Architectures, called Heterogeneous-ISA Chip Multiprocessors. This work leverages such an architecture to realize substantial new security benefits, and in particular, to thwart Return-Oriented Programming. This paper proposes a novel security defense called HIPStR – Heterogeneous-ISA Program State Relocation – that performs dynamic randomization of run-time program state, both within and across ISAs. This technique outperforms the state-of-the-art just-in-time code reuse (JIT-ROP) defense by an average of 15.6%, while simultaneously providing greater security guarantees against classic return-into-libc, ROP, JOP, brute force, JIT-ROP, and several evasive variants.

1. Introduction

Heterogeneous chip multiprocessors employ CPU cores of different organization or size that offer varying degrees of micro-architectural complexity [1, 2, 3, 4] and/or core specialization [5, 6, 7, 8]. 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 [9, 10, 11]. While early work on on-chip heterogeneity [12, 13] restricted cores to implement a single instruction set architecture (ISA), recent findings [14, 15, 16, 17, 18] indicate that a heterogeneous-ISA chip multiprocessor (CMP) is not only a viable option, but has greater potential both in terms of performance and efficiency.

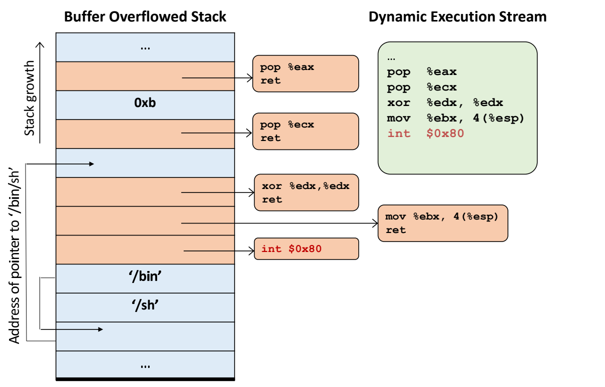
A heterogeneous-ISA CMP synergistically complements architectural heterogeneity with micro-architectural heterogeneity, and allows an application (compiled to each ISA as a fat binary) to dynamically identify the ISA of its preference and migrate execution at any given point of time. In this paper, we 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 (ROP) [19, 20].

Return-oriented Programming chains together short code snippets in the program (called gadgets) that end with a return or an indirect jump instruction, by overflowing the stack with a carefully constructed sequence of return addresses, and other data required for malicious computation. ROP has been shown to be Turing-complete for multiple ISAs, and over a wide range of applications [19, 21, 22, 23, 24, 25]. Several exploit mitigation techniques have been described in the literature to thwart ROP. These mitigations can be broadly classified as (a) control flow integrity (CFI) techniques [26, 27, 28, 29, 30, 31, 32, 33] that constrain execution to a predefined control flow graph, or (b) randomization techniques [34, 35, 36, 37, 38, 39, 40, 41] that enable a system to exist in one of many random states such that it is hard to predict the exact location or manifestation of a gadget.

The success of randomization techniques is directly proportional to the *entropy* (number of randomizable states) they provide, and the extent to which they are resistant to entropy reduction attacks [42, 43, 44]. In their most powerful form, entropy reduction attacks called just-in-time return-oriented programming (JIT-ROP) [45], completely bypass all randomization, using a single leaked memory disclosure. Therefore, it is critical to design robust and performance-efficient randomization techniques that provide an entropy that is beyond the reach of state-of-the-art exploit generation. In this work, we find that the low overhead of execution migration in a heterogeneous-ISA CMP makes it a natural candidate to repel such attacks, and therefore propose a novel defense mechanism called Heterogeneous-ISA Program State Relocation (HIPStR), that performs dynamic randomization of run-time program state, both within and across ISAs.

First, we leverage a heterogeneous-ISA CMP composed of an ARM core and an x86 core, and non-deterministically migrate execution of a vulnerable process between the two ISAs, in such a way that we render JIT-ROP attacks extremely hard to execute, while still retaining inherent performance gains offered by the architecture. Consequently, we remove one of the last remaining "constants" available

ASPLOS '16, April 02-06, 2016, Atlanta, GA, USA
Copyright © 2016 ACM 978-1-4503-4091-5/16/04…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2872362.2872408



```
pop %eax
ret
```

```
pop %ecx
ret
```

```
xor %edx,%edx
ret
```

```
mov %ebx, 4(%esp)
ret
```

```
int $0x80
```

```
...
pop  %eax
pop  %ecx
xor  %edx, %edx
mov  %ebx, 4(%esp)
int  $0x80
```

Figure 1. Return-oriented Programming

to the attacker – knowledge of the ISA the program is executing on.

Second, we note that any program, including a return-oriented program, requires a certain amount of program state, in the form of registers and memory, to perform any computation in the target ISA. To this end, we employ a dynamic binary translation engine on both cores that moves the run-time program state of an application to random and attacker-unknown locations, such that legitimate execution that preserves control flow is guaranteed to function as expected, but an attacker-crafted malicious exploit is highly unlikely to function as intended.

To demonstrate the full potential of the proposed security defense, we subject it to a series of malicious ROP-style attacks, including classic return-into-libc, jump-oriented programming, simple brute force, and just-in-time code reuse. Consequently, we make the following major observations:

(1) The sheer amount of entropy provided by our defense renders brute force attacks such as Blind-ROP [46] practically impossible, for current or even distant future microprocessors.

(2) Our ability to perform seamless and instantaneous execution migration across heterogeneous ISAs significantly inhibits just-in-time code reuse attacks, forcing an attacker to construct heterogeneous-ISA exploits that are extremely hard to execute, as shown in Section 7.

(3) Our performance focused migration policy and our optimization techniques help us outperform Isomeron [35], the only other JIT-ROP defense, by an average of 15.6%, while simultaneously providing greater security guarantees against brute force, JIT-ROP, and more tailored attacks geared to break execution path diversification.

(4) We defend against all variants of classic ROP-style attacks [19, 21, 47, 48], and reduce their overall attack surface to such an extent that it is difficult to construct a four-gadget shellcode exploit, let alone achieve Turing-completeness.

2. Background and Related Work

Heterogeneous-ISA CMPs. Architects have established that on-chip heterogeneity is an effective mechanism to improve processor efficiency for both general purpose and embedded computing [2, 3, 8, 12, 13, 49, 50]. In the embedded world, it is not uncommon for architects to exploit both architectural and microarchitectural heterogeneity to realize heterogeneous-ISA MPSoCs that cater to a diverse class of applications such as wireless networks, signal processing, multimedia, packet switching, and cybersecurity [7, 51, 52, 53, 54]. While examples of heterogeneous-ISA chip multiprocessors do exist in the general-purpose market [6, 8], they haven't been targeted for mainstream mixed workloads yet. However, recent research proposals have demonstrated the performance and energy benefits of such architectures [14, 15, 16, 17, 18]. This work showcases the high potential of this architecture from a security standpoint.

Return-Oriented Programming. Buffer overflow vulnerabilities form a major chunk of the security loopholes that plague the Internet today. They rank third amongst the common vulnerability types reported by the National Vulnerability Database, finishing just behind cross site and cryptographic vulnerabilities [55]. These vulnerabilities have been systematically exploited by code reuse attacks such as *return-into-libc* [48], which exploits a buffer overflow on the stack to return into a *libc* function. Despite its ability to subvert control flow without injecting malicious code, return-into-libc is inherently limited to *libc*, and thus incapable of performing arbitrary malicious computation. In recent years, return-into-libc attacks have evolved into a more general and flexible scheme of attacks called Return-oriented Programming (ROP) [19, 20].

Figure 1 shows a ROP attack that spawns a command shell. The attack begins with an attacker injecting an exploit payload on to the stack, exploiting a buffer overflow. The payload is crafted to overwrite the return address with the address of a short code snippet within the program, called a *gadget*, that ends in a return instruction. Once the gadget has executed and the instruction pointer has reached the return instruction, the stack pointer points to the address of the next gadget, and the exploit continues. ROP hinges on the attacker being able to control the stack pointer and use it as the instruction pointer. Several evasive variants of ROP have been described in the literature that use indirect jumps in place of returns (Jump-oriented programming (JOP) [21, 23, 56]), and provide Turing-completeness on different instruction set architectures [22, 25]. The advent of automated exploit compilers has further made ROP a formidable attack technique to defend against [57].

Control Flow Integrity. Abadi, et al. [26] first formalized the idea of CFI. The main idea of that work is to constrain the execution of the program to a predefined control flow graph (CFG) by instrumenting the program to perform ID-checks before every indirect jump. They also observe that it is difficult to implement ideal CFI statically without a runtime mechanism to track function calls and indirect jumps. There has been significant follow-up work on CFI at the hardware, runtime, and compiler levels [27, 28, 58, 59, 60, 61, 62]. More recent work such as CCFIR [32], bin-CFI [31], branch regulation [30], and code pointer integrity [33] have made significant strides in reducing the attack surface, by employing more fine-grained CFI, in the absence of any source or debug information, and at an acceptable degradation in performance. However, several backdoor attacks [63, 64, 65, 66, 67, 68, 69, 70] have been described in the literature to bypass these techniques, thereby exposing the need for a stricter enforcement of CFI. HIPStR is orthogonal to these defenses and could be applied in conjunction. HIPStR also defends against all the above backdoor attacks since we make no assumptions about CFI or memory safety.

Randomization and Obfuscation. Yet another class of ROP defenses randomize the location of gadgets in the process image, making the attack only probabilistic. Several gadget location randomization techniques have been proposed in the literature, at various granularities — module (ASLR [39]), basic block [41], instruction [36], and byte [40] levels. Furthermore, several binary re-writing and gadget obfuscation mechanisms [37, 38, 71, 72, 73, 74] have been proposed to restrict the number of useful gadgets in a program.

In the presence of a memory disclosure vulnerability, these randomization techniques can be bypassed by simple brute-force attacks [43, 46] that exploit a memory disclosure, in just a matter of a few thousand attempts. Moreover, the load-time nature of these techniques makes them highly susceptible to just-in-time code reuse (JIT-ROP) attacks that exploit a single leaked memory disclosure to read code pages in memory, disassemble them, and reconstruct the control flow graph on-the-fly. Snow, et al. [45] show that JIT-ROP can bypass a combination of fine-grained randomization techniques in a matter of 23 seconds. In this work, we demonstrate significant resistance against entropy exhaustion by employing run-time randomization instead of load-time randomization.

As of this writing, Oxymoron [34] and Isomeron [35] are the only two techniques that have claimed full immunity to JIT-ROP. Oxymoron leverages x86 segmentation and page-level randomization to re-encode direct branches as indirect branches in such a way that the target address of a direct branch is hidden from a JIT-ROP attacker, thereby thwarting the reconstruction of a program's control flow graph on-the-fly. However, Davi, et al. [35] describe a backdoor attack to successfully bypass Oxymoron by disclosing a large number of pages using code pointers, such as return addresses or pointers to virtual methods, gleaned off the stack or the heap.

Isomeron, on the other hand, harnesses software diversity and probabilistic execution by randomly switching between two versions of the program, one original and one diversified, at every function call. Our work differs from Isomeron in that we diversify code both within and across the ISA, thus randomizing the architecture itself. Moreover, HIPStR can defend against return-into-libc and JOP attacks owing to its ability to randomize calling conventions and switch between different ISAs at basic block boundaries, unlike Isomeron which is a function-level diversification technique. HIPStR exploits the already existing architectural heterogeneity in modern processors to provide significantly higher entropy and greater reduction in attack surface, at a lower performance overhead than Isomeron.

JIT-hardening. The NVD database shows that JIT environments such as browsers and Adobe Flash are common targets of code reuse attacks. Specifically, 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 [75]. Such environments typically employ JIT-hardening techniques such as page randomization, constant blinding [75], and random NOP insertion on each JIT-compiled page to defend against JIT-spraying. Our technique can be seamlessly integrated into such JIT environments since we already employ dynamic binary translation. Furthermore, unlike Isomeron, we not only defend against JIT-ROP, but also defend against JIT-spraying, because we enforce program state relocation on all JIT-compiled code including JIT-sprayed gadgets.

3. Architectural Overview

In this section, we lay out our security and performance guarantees, briefly recap the multi-ISA compilation and dynamic task migration methodology outlined in prior work [14, 15], and discuss strategies to harness and re-purpose these techniques as a security defense for ROP.

3.1 Security and Performance Guarantees

Security. One of the main goals of HIPStR is to defend against and reduce the attack surface of a wide array of attacks, including but not limited to return-into-libc, ROP, JOP, brute force attacks, JIT-ROP, and JIT-spraying. For any program in execution, HIPStR dynamically randomizes the location of its program state (registers and stack objects) in order to render brute-force attacks infeasible. Furthermore, HIPStR has the ability to detect a potential break-in attempt via JIT-ROP, and when detected, probabilistically migrates execution to a different ISA, thereby imposing serious limitations on JIT-ROP attacks.

Performance. HIPStR makes several careful performance-related decisions in order to provide security guarantees, at an acceptable degradation in performance, and outperform Isomeron, the only other JIT-ROP defense in the literature. First, unlike Isomeron which diversifies execution at every function call and return, HIPStR migrates execution to a different ISA only when a potential security breach is detected, thereby enjoying full security benefits at virtually zero performance overhead due to migration. Second, HIPStR implements several optimizations described in Section 5 in order to speed up the underlying dynamic binary translation framework.

3.2 Multi-ISA compilation

Process migration across heterogeneous ISAs requires expensive program state transformation because the run-time

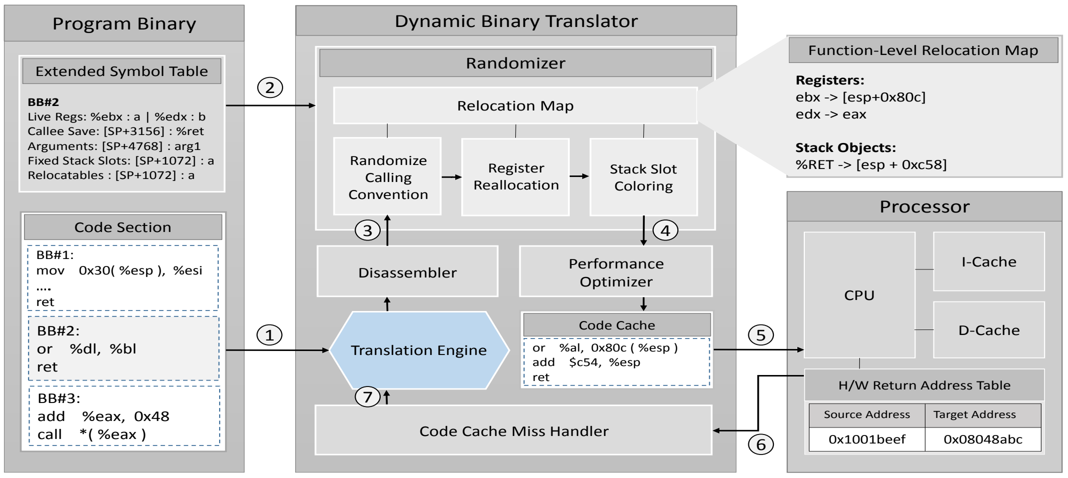


Figure 2. Program State Relocation Architecture

program state (registers and memory) is always present in architecture-specific form. The multi-ISA compilation infrastructure outlined in prior work ensures minimal state transformation by taking advantage of a *symmetrical fat binary* with multiple ISA-specific code sections, a common stack frame organization, and a common ISA-agnostic data section. Furthermore, a multi-ISA runtime transforms the program state (registers and stack) upon migration to a different ISA at specific *equivalence points*, after which point execution can be safely resumed on a different ISA.

3.3 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 require 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 JIT-ROP [45] 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.

3.4 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 2, 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. In Section 5, we describe each transformation in detail. The figure shows an example PSR transformation – BB#2 in the code section is transformed to the basic block shown in the code cache. Note that PSR just requires a simple change in the addressing mode of the instruction *or* in order to relocate its original operands *dl* and *bl* to *al* and *[esp+0x80c]*, as indicated by the relocation map respectively. Note that the return address is also moved to a random location on the stack.

As with classic DBT [76, 77], 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 [46] 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.

3.5 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”(HIPStR).

The defense leverages, in this particular implementation, 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 profits from 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.

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.

4. Assumptions and Threat Model

JIT Engine. We model a JIT engine such as a browser environment that performs dynamic binary transformations. Like most browser environments and other code randomization defenses [35, 36] that employ dynamic binary instrumentation, we assume that the JIT engine is checked for vulnerabilities and its address space is protected by memory protection mechanisms such as code signing [78], sandboxing [79], and Intel Software Guard Extensions (SGX) [80].

Fine-grained Randomization. We require no fine-grained randomization techniques (including ASLR) to protect our system, although they would only further strengthen the system since HIPStR is orthogonal to most existing defense mechanisms.

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 [20] to just-in-time code reuse (JIT-ROP) [45] attacks.

Just-in-time Code Reuse. In addition to the ability to snoop into a program’s memory, 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.

Brute Force Attacks. We also assume that the system is susceptible to brute force attacks such as Blind-ROP [46]. To this end, we model a system as described by Shacham, et al. [43] 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.

5. Design and Implementation

In this section, we present the design and implementation details of Program State Relocation, discuss how our system behaves under different execution scenarios, and finally describe techniques to optimize our system for performance.

5.1 Program State Relocation

As discussed in Section 3, 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.

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.

Procedure Call Transformation. The PSR virtual machine instruments all procedure 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.

Indirect Control Transfer. Like any DBT system [76, 77], the PSR virtual machine traps all indirect jumps into the translator. This ensures 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 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 phase change 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 [15]. 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 78% 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.

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.

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 holds for jump-oriented programming, v-table hijack, and other variants. PSR inherently defeats return-into-libc because of the randomized calling conventions.

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.

5.4 Performance Optimizations

Machine Block Placement. As with any JIT engine, we take measures to carefully place translated basic blocks in the code cache, so that we incur as few conflict misses in the instruction cache as possible. To further improve the instruction cache performance and fetch bandwidth, we align tight single-entry single-exit loops to cache block boundaries.

Branch Inlining and Superblock Formation. Next, we compose our translated basic blocks into *superblocks* that have a single entry-point, but multiple exit-points. We form superblocks in two steps. First, we fold branches whenever possible. This includes both direct unconditional branches and fall-through cases in conditional branches. Second, we avoid backward branches by inlining a direct branch instruction. Note that this results in code duplication, but it both improves the locality in the instruction cache, and reduces pressure on the branch predictor.

Global Register Cache. While PSR provides extremely high entropy, the sheer number of stack operations could potentially cause severe performance degradation. To optimize for performance, we use a global register cache that holds the most frequently used registers that are relocated to stack objects. We mandate that this cache be only three entries long so that we provide high performance in tight loops, and at the same time provide security guarantees by still spilling frequently to random locations.

PSR with a Register Bias. In this final optimization, we perform PSR with a *register bias*, i.e, at all times, we ensure at least three registers are always relocated to other registers, albeit randomized for each function.

5.5 Prototype Implementation of PSR

Owing to the complex addressing modes in x86 and the possibility of *unintentional* gadgets (unaligned sequence of bytes that end with the byte *c3* indicating a *ret* opcode), x86 not only exhibits greater susceptibility to vulnerabilities, but also presents greater challenges in terms of design and implementation. Specifically, we note that the attack space on ARM is 52X smaller than x86 (measured using Galileo [20] ported to ARM), since ARM enforces strict alignment of instructions. Moreover, the simplicity of the instruction set and lower register pressure on ARM facilitate a smaller engineering effort and better opportunity for performance optimization. Therefore, we choose to implement our more complete PSR prototype, and do most of our PSR measurements, in x86. By doing so, we not only demonstrate high coverage (as the vast majority of the gadgets exist in the x86 code), but also report conservative estimates in both performance and security evaluation.

6. Methodology

Security Evaluation. An important characteristic of a security 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 reduce the attack surface (both in terms of availability and functionality), in order to limit the attacker's abil-

Algorithm 1 Brute Force Simulation

```
1: $R = \{r_1, r_2 ... r_m\}$ /* Set of $m$ registers to load. */
2: $P = \emptyset$ /* Set of successfully populated registers. */
3: $X = ()$ /* List of chosen gadgets for the attack. */
4: $Y = ()$ /* List of return address locations for chosen gadgets. */
5: $A(g)$ is the randomized return address for gadget $g$

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

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

ARM core			
Frequency	2 GHz	I cache	32 KB, 2 way
Fetch width	2	D cache	32 KB, 2 way
Issue width	4	ROB size	20 entries
LQ/SQ size	16/16 entries	Functional Units	Int ALU(2), IntMult/Div(1), FP ALU/Mult/Div(2)
x86 core			
Frequency	3.3 GHz	I cache	32 KB, 2 way
Fetch width	4	D cache	32 KB, 2 way
Issue width	4	ROB size	128 entries
LQ/SQ size	48/96 entries	Functional Units	Int ALU(6), Mult/Div(1), FP ALU/Mult/Div(2), SIMD(2)

Table 1. Architecture detail for ARM and x86 cores

ity to construct meaningful exploits. In our evaluation, we not only subject our defense to state-of-the-art attack mechanisms [19, 21, 45, 47, 48], but also measure its effectiveness against potential attacks that are computationally beyond the reach of today's attacker. For each attack, we report the degree to which the attack surface is reduced by our technique.

We use the 'Galileo' algorithm described by Shacham, et al. [20] to mine a benchmark for every possible instruction sequence that ends with a return instruction. 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 an attacker supplied value from the stack as *viable*. We evaluate every gadget for its viability on a system, without and with PSR, to measure the attack surface for four major classes of attacks: (a) classic ROP, (b) brute-force, (c) JIT-ROP, and (d) tailored heterogeneous-ISA attacks to defeat HIPStR.

Owing to the sheer number of stack locations available for program state relocation, the number of possible manifestations of a gadget explodes. 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. [81] showed that the shortest aligned gadget chain generated by gadget compilers such as Q [57] 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, our attack *sprays* 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 (by an earlier gadget in the exploit) state. The algorithm stops searching for more *viable* gadgets as soon it finds a four-gadget shellcode exploit.

Performance Evaluation. 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 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 [82] architectural simulator. The processor model of the ARM core is based on the low-power Cortex A-

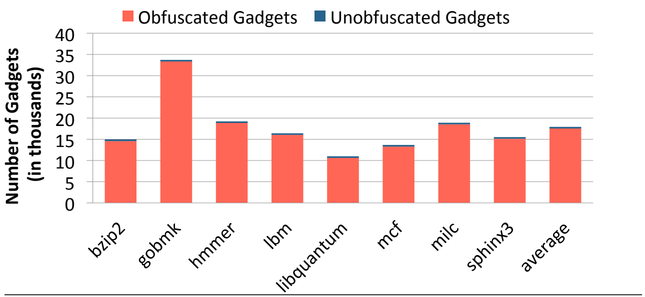


Figure 3. Classic ROP Attack Surface

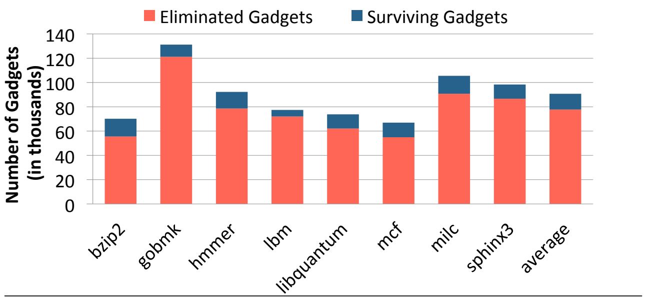


Figure 4. Brute Force Attack Surface

9, while the x86 core is modeled after the high performance Intel Xeon. Table 1 shows the details of each core.

To evaluate the steady state performance and study the effect of various contributing factors, we simulate a portion of the program's execution at different optimization and entropy levels, with different code cache and hardware Return Address Table (RAT) sizes, as follows. We fast forward execution for the first one billion instructions to skip initialization code, and perform cycle accurate simulation for another one billion instructions[83], while running in the context of a PSR virtual machine and with heterogeneous-ISA migrations enabled.

To evaluate the migration overhead at random execution points, we skip the initalization phase and fast forward execution of each benchmark to a random checkpoint and force migration to a different ISA, and report results averaged across ten random checkpoints.

7. Evaluation

7.1 Security Evaluation

Classic ROP-Style Attacks. Figure 3 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 a very small portion of the attack surface remains unaltered. To be precise, PSR reduces the attack surface of classic ROP by an average of 98.04%. We note that although the remaining 1.96% is unobfuscated by PSR, the attacker has no way of determining which gadgets they are beforehand, since their randomized version is only generated on execution, thereby rendering classic ROP attacks infeasible.

Benchmark	Randomizable Params (avg)	Entropy	Attempts (no reg-bias)	Attempts (reg-bias)
bzip2	6.76	88	9.11×10^{33}	2.34×10^{33}
gobmk	6.53	85	2.34×10^{34}	2.87×10^{34}
hmmer	6.69	87	1.37×10^{34}	1.16×10^{34}
lbm	6.92	90	3.33×10^{34}	3.90×10^{34}
libquantum	6.76	88	1.05×10^{34}	6.45×10^{33}
mcf	6.69	87	3.10×10^{33}	1.71×10^{34}
milc	6.46	84	1.86×10^{34}	2.92×10^{34}
sphinx3	6.92	90	1.14×10^{34}	8.68×10^{33}

Table 2. Inferences from Brute Force Simulation

Brute Force Attacks. As illustrated in Figure 3, PSR modifies a majority of the gadgets that were previously available for ROP. These gadgets, by virtue of PSR's transformations, have either been obfuscated in a way that they no longer perform the attacker intended action, or have been completely eliminated. The former of these are *viable* candidates for a brute force attack since they perform useful computation, just not what an attacker expects them to. Even under the assumption of full memory disclosure, it is impossible to determine the transformations that will be applied on a gadget, without executing it. Also *viable* for a brute force attack are any gadgets introduced by the randomization itself. 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. As shown in Figure 4, we observe that a sizable portion (an average of 15.83%) of all gadgets are *viable* for brute force, and therefore require thorough evaluation.

Table 2 shows the results of our brute force simulation described in Section 6. 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 randomizable 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.

Just-In-Time Code Reuse. Figure 5 shows the reduction in attack surface for each benchmark under both single-ISA and heterogeneous-ISA PSR. 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. We find that the number of gadgets already randomized by PSR accounts for only 1.45% of all classic ROP gadgets and 1.92% of those *viable* for brute force, thereby severely constraining the attack surface. Note that a majority of gadgets are now *undiscoverable*, since they lie outside the code cache.

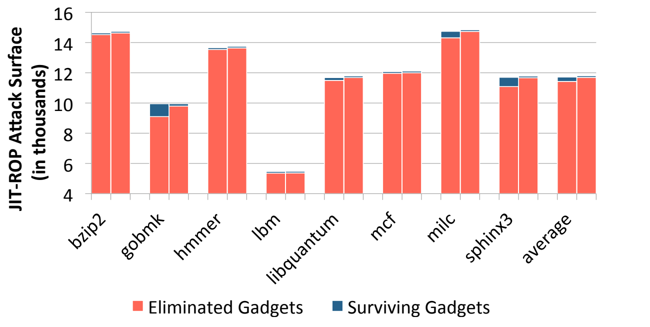


Figure 5. JIT-ROP Attack Surface on (a) PSR, (b) HIPStR

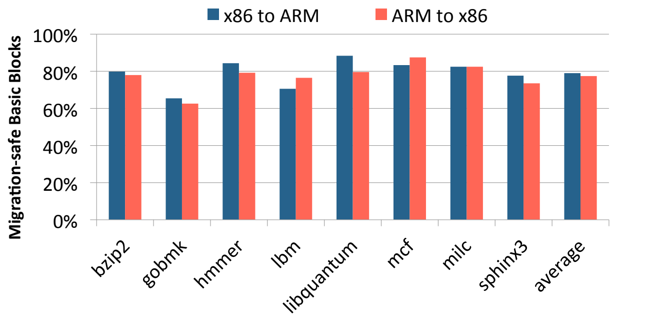


Figure 6. Percentage of Migration-Safe Basic Blocks

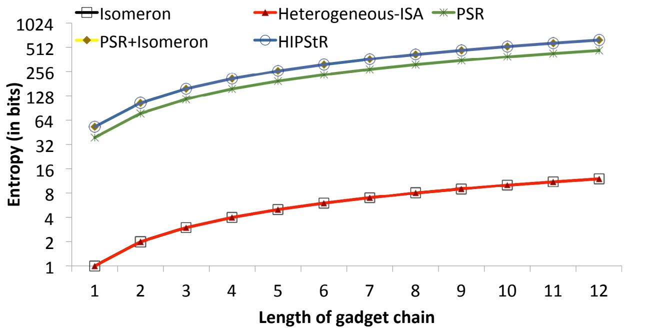


Figure 7. Entropy Comparison

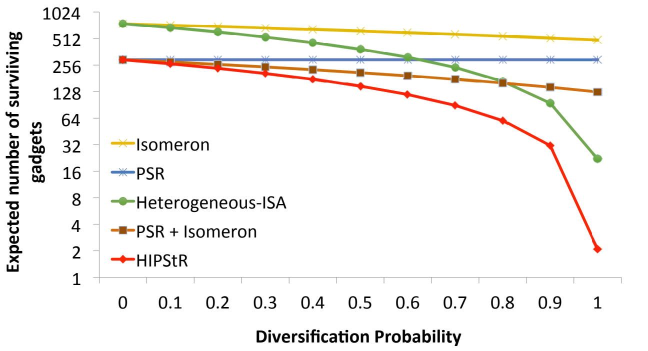


Figure 8. Effect of diversification on attack surface

Although the attack surface has been considerably reduced, the surviving 294 gadgets could potentially be enough to mount a JIT-ROP attack. Recall from Section 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 surviving gadget that is *viable* for JIT-ROP must ideally 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.

We find that out of the 294 surviving gadgets from PSR, 267 gadgets cause a security breach violation in the PSR virtual machine, thereby triggering a probabilistic migration to a different ISA. This leaves the attacker with only 27 gadgets, on average, that do not flag a violation and could potentially bypass migration to a different ISA. Furthermore, as shown in Figure 6, 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.

Tailored attacks. To further explore the synergy of the HIPStR components, we compare the combined entropy of HIPStR with the two individual components of HIPStR (PSR and heterogeneous-ISA migration) alone, as well as a hybrid of Isomeron and our PSR approach (See Figure 7). We make two important observations. First, any system that implements only Isomeron or only Heterogeneous-ISA migration suffers from extremely low entropy, which cannot be amortized unless the gadget chain is long enough. For example, every one out of as low as 256 attempts will succeed for a gadget chain that is 8 gadgets long. 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. Although PSR by itself is susceptible to JIT-ROP, this characteristic of PSR makes brute forcing a JIT-ROP attack significantly harder on a system that implements both diversification and PSR.

This might suggest that a system implementing a combination of PSR and Isomeron is as effective as HIPStR. However, we note that entropy as a metric by itself does not completely capture the effect of randomizing the instruction sets. To observe this effect, we turn to tailored attacks that bypass both same-ISA (Isomeron) and heterogeneous-ISA diversification. An attacker who is aware of the diversification could construct exploits that interleave gadgets from both the original and the diversified versions. For example, on HIPStR, one could craft an exploit that alternates gadgets between x86 and ARM, such that the all meaningul computation is performed by x86 gadgets, while the ARM gadgets are all *nops* that switch execution to x86 without clobbering already

Level	Optimizations
-O0	No Optimization.
-O1	Machine Block Placement, Branch Inlining and Superblock Formation.
-O2	-O1 optimizations, Global Register Cache.
-O3	-O2 optimizations, PSR with a register bias.

Table 3. Performance Optimizations for PSR

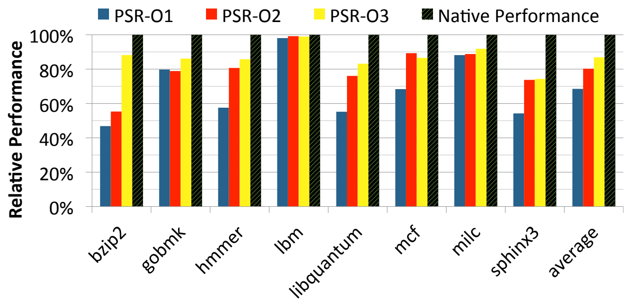


Figure 9. Performance at different optimization levels

established state. Another example of such a tailored attack would be to use those gadgets that are unaffected by diversification, i.e, those gadgets that perform the same intended malicious operation regardless of what ISA/software version they execute.

Figure 8 examines the JIT-ROP attack surface of each technique in the face of such tailored attack mechanisms. When the probability of diversification (the probablility of switching ISAs or program variants between gadgets) is zero, such an attack surface will include all gadgets present in the code cache. As the diversification probability increases, the attacker would find it increasingly harder to brute force a JIT-ROP attack.

We note that the system implementing both PSR and Isomeron is as effective as HIPStR when the diversification probability is zero, but the two systems rapidly diverge in their effectiveness as the probability increases. In fact, at a probability of one, at which we diversify execution for every gadget, HIPStR manages to have an average of just two surviving gadgets in its attack surface, while the attack surface of the system implementing PSR and Isomeron still comprises hundreds of gadgets. It is worth noting here that on HIPStR, we failed to find any surviving gadgets in five out of the eight applications we benchmark, thereby completely thwarting such tailored attacks.

Aside from these eight SPEC applications, we also evaluate the effectiveness of HIPStR on the network facing daemon *httpd*, a classic target of ROP attacks. Our evaluation shows that the attack surface of *httpd* is composed of 169,272 gadgets. PSR successfully obfuscates 99.7% of the gadgets, requiring 1.8x10^{39} attempts to brute-force (still computationally infeasible). Furthermore, while 84 gadgets are available for JIT-ROP, only two survive heterogeneous-ISA migration. These are insufficient to generate even the simplest shellcode exploit.

In our evaluation, we find that it is more likely to find large gadgets that populate multiple registers at a time, and

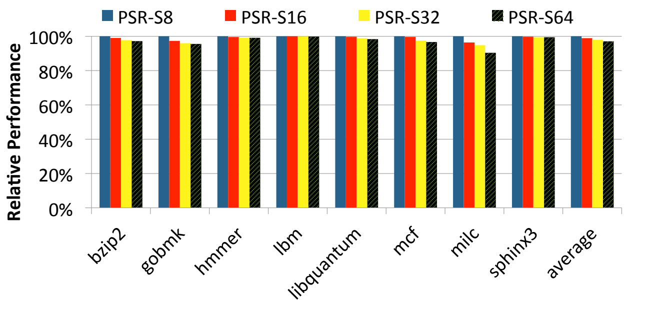


Figure 10. Effect of additional stack memory overhead

are unaffected by diversification on the same ISA, rather than different ISAs. Note that our experiments only measure the number of surviving gadgets in the face of such tailored attacks. We expect that chaining together these gadgets to craft an exploit payload is a much more daunting task, because not only is brute-forcing such an attack under PSR extremely hard, but heterogeneous-ISA migration involves stack transformation that could potentially clobber the exploit payload on the stack.

The strength of PSR lies in its ability to defeat brute force attacks, while simultaneously reducing the attack surface. It amplifies the entropy of heterogeneous-ISA migration making a brute force attack on the combined defense infeasible. Conversely, heterogeneous-ISA migration has the ability to shield PSR from a JIT-ROP attack attempting to bypass randomization. Together, they form a formidable defense.

7.2 Performance Evaluation

Steady State Performance. Figure 9 shows the steady state performance overhead of PSR and the effect of each performance optimization. We do not gain a significant performance boost from the O1 level of optimizations that we apply to improve the instruction cache performance. However, we observe an average of 13% improvement in performance due to a small global register cache that holds just 3 registers. This, in large part, is due to the fact that short and tight loops operate on a small set of input registers, but dominate most of a program's execution. Finally, we find that operating PSR in a register-bias mode can further improve performance by an average of 5.5%, thereby reducing the overall performance degradation from native execution to just 13.14%.

Since PSR relies on large stack frames to provide higher entropy, it is important to measure the performance effects due to the extra stack memory used. Figure 10 shows the steady state performance overhead at different entropy levels. The performance only drops by an average of 2.96% even after expanding individual stack frame sizes by as much as 64KB. This is because the stack frames become very sparse, and large empty spaces between items (e.g., larger than a cache line) do not place pressure on the cache.

PSR takes advantage of a return address table (Section 3) to protect the code cache from becoming compromised. Figure 11 shows the effect of the RAT size on performance. We

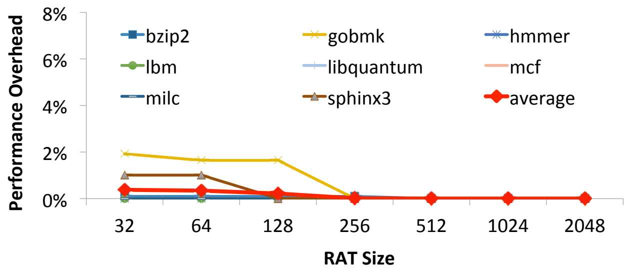


Figure 11. Effect of RAT size on Performance

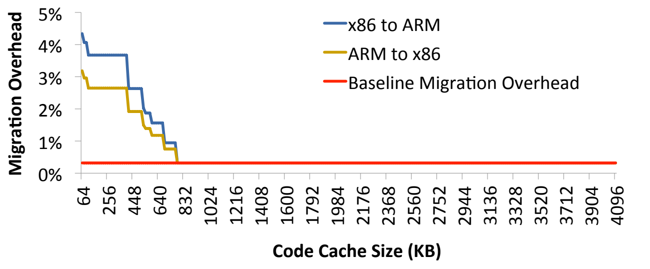


Figure 13. Effect of code cache size on Performance

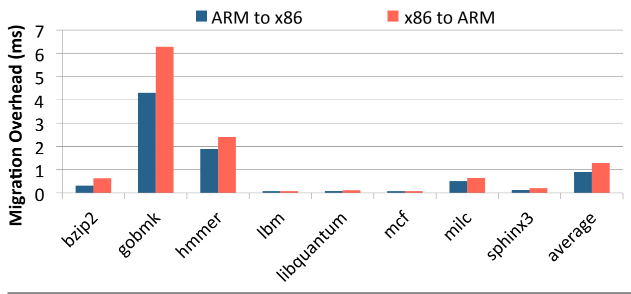


Figure 12. Migration Overhead

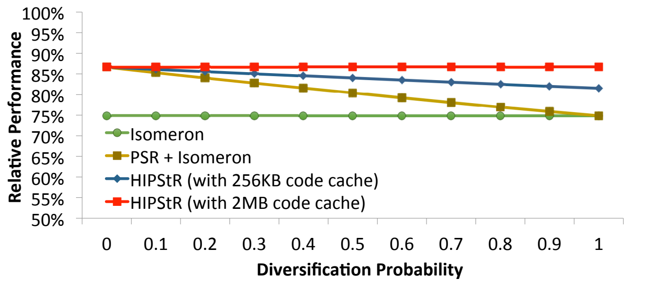


Figure 14. Performance Comparison with Isomeron

incur an average of 0.37% performance overhead even with as small as a 32-entry RAT. In fact, we observe no noticeable degradation in performance with a RAT that can hold just 512 entries. This is because the distance between a *call* and a *return* instruction is generally so short that we seldom incur a RAT miss.

Migration Overhead. Figure 12 shows the state transformation overhead due to heterogeneous-ISA process migration. We report an average migration overhead of 909 microseconds when migrating from ARM to x86, and 1.287 ms in the other direction, resulting in an overall baseline migration overhead of 0.32% resulting from the migrations initiated to improve performance. Our migration policy ensures that we switch ISAs only when a program's ISA preference changes as it enters a new program phase, or when the PSR virtual machine suspects a security breach, i.e, when we encounter an indirect control transfer that results in a code cache miss.

Figure 13 shows the effect of code cache size on migration overhead. We record zero indirect control transfers that miss a code cache as small as 768 KB, resulting in no measurable overhead for security-induced migrations. In steady state execution, it is highly unlikely that we return to a function whose translation has been evicted, or we make an indirect jump or a function pointer call to an evicted region. For example, *gobmk* makes 65,746 function pointer calls within a span of one second, but none miss the code cache.

Finally, Figure 14 compares the performance of HIPStR with Isomeron, the only other technique that defends against JIT-ROP, along with a system that implements both PSR and Isomeron, for the six common applications that we benchmark. Since Isomeron invokes execution path diversification for each function call and return, they report a higher performance overhead. In fact, they mention that this degradation in performance is expected since their program shepherding renders CPU optimizations like branch prediction ineffective [35]. As the diversification probability increases, HIPStR performance only slightly decreases with the smaller code cache, but still manages to provide higher performance than both Isomeron, as well as the combination of PSR and Isomeron. Overall, HIPStR outperforms Isomeron by an average of 15.6% while simultaneously providing signficantly higher entropy, and thereby higher resistance to brute force, JIT-ROP, and tailored attacks that bypass diversification.

8. Conclusion

Heterogeneous-ISA CMPs have been shown to provide significant performance and energy gains. In this work, we showcase their security benefits through a novel defense called HIPStR (Heterogeneous-ISA Program State Relocation) that combines dynamic randomization of program state with non-deterministic process migration between heterogeneous-ISAs. We find that HIPStR outperforms the state-of-the-art JIT-ROP defense by 15.6% while providing greater security guarantees.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful insights. The authors are particularly grateful to Arvind Krishnaswamy, Koichi Yamada, and Rajan Palanivel for their numerous suggestions during the early days of PSR's formulation, and for their contributions to the United States Patent US009135435B2 [84]. This research was supported in part by NSF Grant CCF-1302682.

References

[1] R. Kumar, D. M. Tullsen, N. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, 2005.

[2] "Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance," tech. rep., NVidia, 2011.

[3] P. Greenhalgh, "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," tech. rep., ARM, 2011.

[4] M. Hill and M. Marty, "Amdahl's Law in the Multicore Era," *Computer*, July 2008.

[5] "2nd Generation Intel Core vPro Processor Family," tech. rep., Intel, 2008.

[6] "The future is fusion: The Industry-Changing Impact of Accelerated Computing.," tech. rep., AMD, 2008.

[7] "The Benefits of Multiple CPU Cores in Mobile Devices," tech. rep., NVidia, 2010.

[8] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, July 2005.

[9] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE computer*, 2007.

[10] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceedings of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[11] G. Varsamopoulos, Z. Abbasi, and S. K. Gupta, "Trends and effects of energy proportionality on server provisioning in data centers," in *Proceedings of the 17th Annual International Conference on High Performance Computing*, 2010.

[12] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction," in *International Symposium on Microarchitecture*, Dec. 2003.

[13] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance," in *International Symposium on Computer Architecture*, June 2004.

[14] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution migration in a heterogeneous-isa chip multiprocessor," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[15] A. Venkat and D. M. Tullsen, "Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor," in *Proceedings of the International Symposium on Computer Architecture*, 2014.

[16] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms," in *Proceedings of the 10th European Conference on Computer Systems*, Apr. 2015.

[17] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-ISA heterogeneous multi-core architectures," in *Proceedings of the 16th International Symposium on High Performance Computer Architecture*, Jan. 2010.

[18] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures," in *Proceedings of the 42nd International Symposium on Computer Architecture*, June 2015.

[19] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security*, 2012.

[20] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and Communications Security*, 2007.

[21] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[22] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to RISC," in *Proceedings of the 15th ACM conference on Computer and Communications Security*, 2008.

[23] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and Communications Security*, 2010.

[24] S. Checkoway and E. W. Felten, "Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage," 2009.

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

[26] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and Communications Security*, 2005.

[27] C. Cowan, C. Pu, D. Maier, *et al.*, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium*, 1998.

[28] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[29] H. Etoh, "GCC extension for protecting applications from stack-smashing attacks," 2003.

[30] M. Kayaalp, M. Ozsoy, N. Abu Ghazaleh, and D. Ponomarev, "Branch regulation: low-overhead protection from code reuse attacks," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[31] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[32] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[33] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[34] Michael Backes and Stefan Nürnberger, "Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing," in *Proceedings of the 23rd USENIX Security Symposium*, Aug 2014.

[35] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," July 2015.

[36] J. Hiser, A. Nguyen Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd My Gadgets Go?," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[37] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and Communications Security*, 2003.

[38] V. Pappas, M. Polychronakis, and A. 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*, 2012.

[39] PaX Team, "PaX address space layout randomization," 2003.

[40] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu, "Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks.," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[41] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and Communications Security*, 2012.

[42] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib (c)," in *Proceedings of the 25th Annual Computer Security Applications Conference*, 2009.

[43] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and Communications Security*, 2004.

[44] B.-J. Wever, "Internet Explorer IFRAME src&name parameter BoF remote compromise," 2004.

[45] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. 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*, 2013.

[46] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking Blind," in *Security and Privacy*, July 2014.

[47] H. D. Moore, "Microsoft Internet Explorer data binding memory corruption," 2010.

[48] Solar Designer, "Getting around non-executable stack (and fix)," 1997.

[49] G. Kyriazia, "Heterogeneous Systems Architecture: A Technical Review," tech. rep., 2012.

[50] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of the 41st International Symposium on Computer Architecture*, June 2014.

[51] D. Allred and G. Martinez, "Maximizing the Power of ARM with DSP," tech. rep., Texas Instruments, 2010.

[52] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A multiprocessor SoC for advanced set-top box and digital TV systems," *Design & Test of Computers, IEEE*, vol. 18, no. 5, 2001.

[53] "Intel IXP425 Network Processor," tech. rep., 2006.

[54] Qualcomm, "Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age," tech. rep., Oct. 2011.

[55] "National Vulnerability Database,"

[56] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks," in *Proceedings of the 21st International Symposium on Network and Distributed System Security*, Feb. 2014.

[57] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy.," in *Proceddings of the 20th USENIX Security Symposium*, 2011.

[58] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis, "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[59] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting systems from stack smashing attacks with StackGuard," in *Proceedings of the 5th Linux Expo*, 1999.

[60] H. Ozdoganoglu, T. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote, "SmashGuard: A hardware solution to prevent security attacks on the function return address," *IEEE Transactions on Computers*, 2006.

[61] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[62] Vendicator, "StackShield: A Stack Smashing Technique Protection Tool for Linux," 2001.

[63] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point (er): On the effectiveness of code pointer integrity1," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[64] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Security Symposium*, 2015.

[65] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, May 2014.

[66] E. Goktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and

G. Portokalidis, “Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard,” in *Proceedings of the 23rd USENIX Security Symposium*, Aug. 2014.

[67] N. Carlini and D. Wagner, “Rop is still dangerous: Breaking modern defenses,” in *Proceedings of the 23rd USENIX Security Symposium*, Aug. 2014.

[68] Lucas Davi, Daniel Lehmann, and Ahmad-Reza Sadeghi, “The Beast is in Your Memory: Return-Oriented Programming Attacks Against Modern Control-Flow Integrity Protection Te chniques,” in *BlackHat USA*, Aug 2014.

[69] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, “Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection,” in *Proceedings of the 23rd USENIX Security Symposium*, Aug. 2014.

[70] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming,” May 2015.

[71] S. Bhatkar and R. Sekar, “Data space randomization,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[72] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, “Data randomization,” tech. rep., Technical Report MSR-TR-2008-120, Microsoft Research, 2008.

[73] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, “G-Free: defeating return-oriented programming through gadget-less binaries,” in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

[74] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis, “ASIST: architectural support for instruction set randomization,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, 2013.

[75] C. Rohlf and Y. Ivnitskiy, “Attacking clientside JIT compilers,” *Black Hat, USA*, 2011.

[76] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., June 2005.

[77] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *USENIX Technical Conference*, Apr. 2005.

[78] MSDN, “Introduction to code signing,”

[79] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, “Language-independent sandboxing of just-in-time compilation and self-modifying code,” *ACM SIGPLAN Notices*, 2011.

[80] Intel, “Software guard extensions programming reference,” 2014.

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

[82] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, “The M5 Simulator: Modeling Networked Systems,” *Micro, IEEE*, 2006.

[83] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically Characterizing Large Scale Program Behavior,” in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[84] A. Venkat, A. Krishnaswamy, K. Yamada, and R. Palanivel, “Binary Translation driven Program State Relocation,” in *United States Patent Grant US009135435B2*, 2015.