Architectural Support for Control-Flow Integrity

George Christou

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
University Campus, Voutes, Heraklion, GR-70013, Greece

Thesis Advisors:
Prof. Evangelos Markatos,
Dr. Sotiris Ioannidis

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

Architectural Support for Control-Flow Integrity

Thesis submitted by
George Christou
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____
George Christou

Committee approvals: _____
Evangelos Markatos
Professor, Thesis Supervisor

_____
Sotiris Ioannidis
Principal Researcher, Committee Member

_____
Angelos Bilas
Professor, Committee Member

Departmental approval: _____
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, 20 February 2017

Abstract

Exploitation of software becomes more and more common, as computer systems span across many areas of our lives. Over the recent years, attacks on software become more sophisticated. Deployed countermeasures tend to not provide sufficient protection. Effective countermeasures require thorough checks which are computantionally expensive.

One such countermeasure is Control-Flow Integrity (CFI); a policy developed to defend against Control-flow hijacking, the principal method for code-reuse techniques like Return-oriented Programming (ROP) and Jump-oriented Programming (JOP). The community proposed CFI, a technique capable of preventing exploitation by verifying that every (indirect) control-flow transfer points to a legitimate address. Enabling CFI in real world systems is not straightforward, since in many cases the actual Control-flow Graph (CFG) of a program can be only approximated. Even in the case that there is perfect knowledge of the CFG, ensuring that all return instructions will return to their actual call sites, without employing a shadow stack, is questionable. On the other hand, the community has expressed concerns related to significant overheads stemming from deploying a shadow stack.

In this work we acknowledge the importance of pushing security in the hardware domain, in order to strengthen and accelerate security mechanisms. We project, that implementing a full-featured CFI-enabled Instruction Set Architecture (ISA) in actual hardware with an in-chip secure memory can be efficiently carried out and the prototype experiences negligible overheads. For supporting our case, we implement Control-Flow Integrity Extensions (CFIX) by modifying a SPARC SoC and evaluate the prototype on an FPGA board by running SPECInt benchmarks instrumented with a fine-grained CFI policy. The evaluation shows that CFIX can effectively protect applications from code-reuse attacks, while adding less than 1% runtime overhead and 2% power consumption overhead, making it particularly suitable for embedded systems.

Περίληψη

Η κακόβουλη εκμετάλλευση λογισμικού γίνεται ολοένα και ποιο συχνή, με την εξάπλωση των υπολογιστικών συστημάτων στην καθημερινότητά μας. Τα τελευταία χρόνια, οι επιθέσεις στο λογισμικό γίνονται ποιο εξελιγμένες. Τα αντίμετρα που υπάρχουν στα υπολογιστικά συστήματα, τείνουν να μην προφέρουν αρκετή προστασία. Τα ισχυρά αντίμετρα απαιτούν ενδελεχής ελέγχους που είναι υπολογιστικά ακριβοί.

Ένα ικανό αντίμετρο είναι η Ακεραιότητα Ροής Εκτέλεσης (CFI). Είναι μια πολιτική που αναπτύχθηκε έτσι ώστε να προστατεύει το λογισμικό από επιθέσεις που αλλοιώνουν τη ροή εκτέλεσής του, τη κύρια μέθοδο για επίτευξη τεχνικών επαναχρησιμοποίησης κώδικα, όπως επιστρεφόμενου προγραμματισμού (ROP) και προγραμματισμού με άλματα (JOP). Η ερευνητική κοινότητα πρότεινε αυτή την τεχνική, ως ικανή να αποτρέπει τις επιθέσεις αυτές, με το να ελέγχει ότι κάθε αλλαγή στη ροή εκτέλεσης ενός προγράμματος και προέρχεται από υπολογισμό της νέας διεύθυνσης, καταλήγει σε σωστή διεύθυνση. Η επέκταση συστημάτων με CFI δεν είναι μια ξεκάθαρη διαδικασία, καθώς ο Γράφος ροής ενός προγράμματος δεν μπορεί να καθοριστεί πάντα με ακρίβεια. Ακόμα και σε περιπτώσεις που ο Γράφος ροής είναι πλήρως καθορισμένος, ο ακριβής έλεγχος ότι οι εντολείς επιστροφής γυρνάνε στη διεύθυνση από όπου έγινε η κλήση, χωρίς τη χρήση μιας προστατευόμενης στοίβας είναι αμφισβητούμενη. Όμως, η ερευνητική κοινότητα αποφεύγει τη χρήση προστατευόμενης στοίβας λόγο της επιβράδυνσης που επιφέρει.

Σε αυτή τη δουλειά αναγνωρίζουμε τη σημαντικότητα της υλοποίησης μηχανισμών ασφαλείας σε επίπεδο υλικού με σκοπό την ενίσχυση και επιτάχυνσή τους. Δείχνουμε, ότι η υλοποίηση μιας αρχιτεκτονικής με εντολές CFI στο υλικό μαζί με προστατευόμενη μνήμη μέσα στον επεξεργαστή είναι εφικτή και το πρωτότυπο είχε ελάχιστη επιβράδυνση. Για να υποστηρίξουμε την ιδέα μας, υλοποιήσαμε τις Επεκτάσεις Ελέγχου Ροής Εκτέλεσης (CFIX), τροποποιώντας ένα σύστημα SPARC και αξιολογήσαμε το πρωτότυπο σε πλακέτα FPGA με το να τρέξουμε SPECint προγράμματα μέτρησης επιδόσεων που είχαν εντολές για λεπτομερή έλεγχο ακεραιότητας ροής. Η αξιολόγηση έδειξε ότι τα CFIX μπορούν να προστατεύσουν αποτελεσματικά τις εφαρμογές από επιθέσεις επαναχρησιμοποίησης κώδικα και παράλληλα επιβραδύνουν το σύστημα κατά 1% και αυξάνουν την κατανάλωση ενέργειας κατά 2%, καθιστώντας το σύστημα μας ιδανικό για ενσωματωμένα συστήματα.

Acknowledgments

I would like to thank my Advisor Dr. Sotiris Ioannidis for his guidance and support during my studies and this work. I would also like to thank my Supervisor, Professor Evangelos Markatos for his overall assistance and interesting discussions all these years. Many thanks to Professor Angelos Billas, for being the third committee member in the evaluation of this work. Special thanks to Nick Christoulakis for implementing together this work and Elias Athanasopoulos for his overall help.

I need to also express my appreciativeness to Evangelos Ladakis, Dimitris Deyannis, Hlias Papadopoulos, Michalis Diamantaris, George Tsirantonakis, Eva Papadogiannaki, Eirini Degleri, Panagiotis Papadopoulos, Kostis Kleftogiorgos, George Vassiliadis, Rafail Tsirmpas ,Lazaros Koromilas, Antonis Krithinakis and Christos Papachristos, my colleagues and most important, friends in the Distributed Computing Systems (DCS) Lab. In like manner, i would like to thank my friends: Nikos, Mimis, Akis, Sotiris, Kostas, Pantelis, Maria Xristina, Kostas, Dimitris, Ioanna, Dr. Fotis, Fotis, Mariellen, Lefas, Flora, George, Katerina, Panito and Eirini. Last but not least, special thanks to Anthi for her constant support and being by my side since the beginning of my master thesis.

Finally, i would also like to heartfully thank my family, my father Christos, my mother Georgia, my sister Ellie and her husband Andreas, my uncle Andreas and my aunt Ioanna and my cousins Miltos and Rafail, for helping me all these years and supporting my choices. This work would have never been possible without them.

This work has been performed at the **Distributed Computing Systems laboratory, Institute of Computer Science, Foundation of Research and Technology – Hellas (FORTH)**. In addition, this work has been supported in part by the European Commission via the H2020 ICT-32-2014 Project SHARCS under Grant 644571.

An early report on this work has been accepted for publication in ACM CODASPY 2016[1], and was awarded with the Outstanding Paper Award.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The current technology trend of introducing *smart* computing capabilities to every day electronic devices, renders our society more vulnerable than ever before to software exploitation. Since many systems are entirely software controlled, they must be protected from adversaries, otherwise the dangers can be very serious. Exploitation of modern software is undoubtedly still possible, despite many mitigation techniques that have been enabled in production systems. More than a decade ago, exploiting software was as easy as just simply smashing the stack [2]. An attacker could fill a vulnerable buffer located in the stack with their code, write past the buffer smashing the stack, and changing the return address (of the current stack frame) to point back to their code. Today, this is not possible anymore due to non-executable data protection (DEP) [3], but attackers can still exploit software. Advanced exploitation techniques, based on code reuse, commonly known as Return-Oriented Programming (ROP) [4] and Jump-Oriented Programming (JOP) [5], are so powerful that can potentially take advantage of any vulnerability and transform it to a fully functional exploit. These techniques do not introduce new code, but *new functionality* in the vulnerable program. Attackers re-use existing parts of the program and build exploits that can work even when DEP is in place. Code randomization techniques [6, 7, 8, 9] attempt to make code reuse harder by shuffling the location of the code to be reused, but it has been demonstrated that even a simple information leak can reveal all of the process' layout and essentially bypass any randomization scheme [10].

Instead of hiding the code, another potential avenue for stopping exploits is to prevent new functionality from executing. One promising direction is based on the observation that modern exploits introduce control-flows that are not part of the program's Control-flow Graph (CFG). Control-flow Integrity (CFI) [11] suggests that a running program should exhibit only the control-flows that are part of the program's original CFG as expressed by its source code. Essentially, CFI mandates that any indirect branch should not be possible to target the address of *any* instruction in the program, but rather be constrained in an allowable set of addresses that have been a priori determined. For example, consider that in principle a return instruction should be *only* able to transfer control to the call site responsible for the associated function call.

CFI, although powerful, still has two open issues related to *accuracy* and *performance*.

As far as accuracy is concerned, it is not always easy to compute the program's CFG. This is mainly because the source code might not be always available, dynamic code might be introduced at run-time [12], and heavy use of function pointers can lead to inconclusive target resolution. This problem has led researchers to develop CFI techniques that are based on a relaxed approximation of the CFG [13], also known as coarse-grained CFI. Unfortunately, coarse-grained CFI has been demonstrated to exhibit weak security guarantees and it is today well established that it can be bypassed [14]. Approximation of the ideal CFG through code analysis is not always sound, therefore, at least for protecting backward edges, the community has suggested *shadow stacks* [15] - secure memory that stores all return address during function calls. Many research efforts have stressed that shadow stacks are important for securing programs, even when we know the program's CFG with high accuracy. As was recently demonstrated even an *ideal* CFI implementation, without the use of a shadow stack, *is* vulnerable [16, 17]. A trivial case is when a function (e.g., `memcmp`) is called by multiple places in the program. According to the CFG, all return locations are legitimate, however only one is actually correct. This, essentially allows the attacker to be flexible in creating chains of call-preceded gadgets for finally exploiting the program. It is thus vital for *any* CFI implementation to employ a shadow stack.

In this work, we acknowledge the enhanced security guarantees and performance boost of hardware implemented security mechanisms. In order to support this acknowledgement we used CFI as the reference use case. CFI is an interesting concept and an active research field in systems security. Although we still lack of a perfect defense policy against all exploits, the community actively seeks new algorithms for realizing CFI flavors. We explore CFI in the context of a hardware implementation. By extending a pre-existing Instruction Set Architecture(ISA) with new instructions dedicated for CFI, and deploying shadow memory inside the processor core, we created CFIX, a full-featured hardware implementation of CFI. We further attempt to quantify the performance overhead of CFI and demonstrate that the technique can be applied to real systems with practically negligible overhead. We evaluate the prototype on an FPGA board by running SPECInt benchmarks instrumented with the additional CFI-related instructions. The evaluation shows that CFIX can effectively protect applications from code-reuse attacks, while adding less than 1% runtime overhead.

Compared to similar hardware implementations, such as HAFIX [18], CFIX is (i) *complete*, since it protects both forward and backward edges, (ii) *faster*, since the experienced overhead is on average less than 1%, and (iii) *more accurate*, since it employs a full-functional shadow stack implemented inside the core. Especially, as far as shadow memory is concerned, CFIX uses a novel system for supporting multiple recursive calls. Each time a return address is to be saved in the secure memory it is checked with the top of the shadow stack and if the address is matched, indicating there is a recursive call, no additional memory is wasted. This dramatically simplifies the design and reduces the space requirements, but implies that a recursive call can return to its call site immediately from any depth, thus violating a perfect CFI policy. However, we anticipate that this policy relaxation has not severe security implications, since system calls and sensitive functions are not recursive and they do not call recursive functions (i.e., hijacking a

recursive function called by a sensitive system call for jumping to the sensitive call site is not possible). Furthermore, in terms of completeness, we argue that CFIX is the most rich hardware implementation of CFI so far, supporting many problematic cases (such as `setjmp`/`longjmp`), which we discuss thoroughly in Section 4.

As new exploit mitigation techniques and technologies arise, attack methods become more and more sophisticated, in order to work around or through them. Proposals for defence are implemented only if they don't hinder performance or functionality, and those already in place rarely offer the security they envisioned. This effectively creates an arms race in which, since new fine grained defence mechanisms are harder to deploy due to excessive runtime overhead and incompatibility with legacy software, the offensive party has the advantage.

1.1 Code injection

Code injection is the exploitation of a system's vulnerability, usually caused by the erroneous handling of input data. Code injections can occur in many different types of applications. For example, in SQL applications text with malicious SQL commands are inserted in the input fields and due to the absence of input sanitization checks, an attacker can modify or read sensitive data from a database. In other cases, adversaries can compromise servers by injecting server scripting code. A common example in php is the use of *eval* function without proper validation of the input can offer an attacker the ability to execute arbitrary code, even system commands, with the same permission as the target web service. In application binaries, low-level code injection can be achieved when buffer overflow vulnerabilities are present in the application's code. During a buffer overflow, a program writes data to a buffer, overruns the boundaries allocated for the buffer and overwrites contiguous memory areas. This behaviour is a well-known security exploit. By sending specially crafted data to the system it is possible to overwrite code sections with malicious code, hijacking the system's functionality. Another method is to write malicious code and direct the execution to it. This can happen by overwriting a return address with the address where the buffer holding the malicious code begins. Since the return and data buffers both reside in the stack a buffer overflow can overwrite the return address with the address the adversary desires. Stack canaries [19] is a technology, used to detect modifications to the stack by placing a unique label before any sensitive data, like the return address, but an attacker can easily bypass this security feature by overwriting it bit by bit and observing the results. In this work, we focus on defending against buffer overflow related exploits.

1.1.1 Modern defences

In modern systems such attacks are not effective, since several counter measures prevent the execution of injected code. Most operating systems are configured with W$\oplus$X policy. Thus, any attempt to direct the execution on a page with data will result in General Protection Fault (GPF). Respectively, the same will happen if due to a buffer overflow a store

operation targets an address inside a page containing code. Many modern processors support a feature called No Execute (NX), which system software utilizes in order to mark pages containing the stack and heap as non-executable.

1.2 Code Reuse

Since Data Execution Prevention arrests the execution of injected malicious code, modern attacks do not depend on injecting malicious code and transferring execution to it. As a result, attackers sifted the focus to using code already present in executable address spaces for their nefarious purposes. Code reuse describes a class of sophisticated security exploits. Herein, an attacker who can overwrite the stack, heap or both of an application, identifies small pieces of code succeeded with indirect branch instructions (e.g. indirect calls and returns) in order chain this pieces and form a full fledged exploit. For example, Return-oriented programming (ROP) is a technique that focuses on hijacking the control-flow of the target program in order to force it outside the normal instruction execution sequence. It affects many common used processor architectures including x86, ARM [20] and SPARC [21] It's a more advanced version of stack-smashing, in that it utilizes vulnerabilities that modify the stack in order to overwrite the return addresses stored within. The new return addresses are then used to change the control-flow of the executable to the attacker's desired path1.1.

Since the attacker now has control over the stack, and subsequently the return addresses, he can jump to any executable memory in the program's address space. The logical next step is to identify the code that he wants to run and set the return address to point to it. The identified pieces of code are called gadgets. Jump-oriented programming (JOP) relies on the same core idea as ROP, but instead gadgets are chained through overwriting function pointers.

1.2.1 Gadgets

Gadgets are small sequences of instructions that typically end with a return instruction. They provide a plethora of functionalities like pushing items to the stack from certain registers, executing arithmetic and logical operations, performing memory operations, etc. They can be used by the attacker to achieve a desired functionality, like setting the environment to execute a call to a certain function.

The attacker chains these gadgets together by pushing the address of each to the stack, through a stack related vulnerability like a buffer overflow. Since gadgets end with a return instruction, at the end of its execution, a gadget will jump to the next address pushed to the stack. The attacker can identify what functionality a gadget will provide either by disassembling the program, or, if the binary is not available for analysis, by observing the effects each gadget has on the stack and the control-flow in general, as demonstrated by Bittau et al. in Hacking Blind[22]. In CISC processors, gadgets are more common, since the attacker can point execution to any byte of an instruction, hence causing the interpretation of an instruction to shift away from its original functionality.

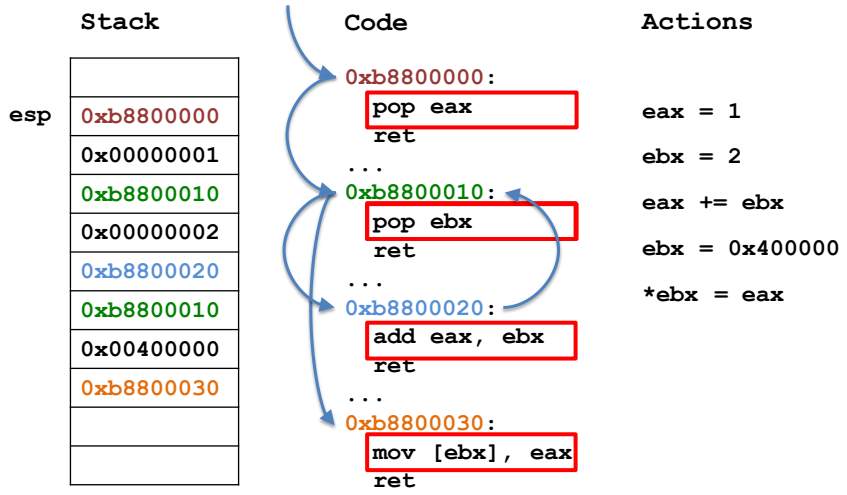


Figure 1.1: Example of how a ROP attack is launched, in the stack has been overwritten in order to chain the gadgets (by overwriting return addresses) and populate the stack with values to be used by the gadgets. On the right is what will be the result of the gadget chain. Figure by V.Pappas.

1.2.2 Return-to-libc

A "return-to-libc" attack relies on overwriting a return address on the call stack, with an address of a subroutine already present in the process' memory. In many programs, a vast majority of system libraries are loaded with the application, giving the attacker a plethora of functions to be used in order to achieve system compromisation.

1.2.3 Modern Defences

By deploying Address Space Layout Randomization (ASLR)[23] this type of attack is unlikely to succeed. ASLR is available in most operating systems (Linux, Windows, Mac OS X). This technique randomly shuffles the position of the major components of an executable such as heap, stack and libraries in the process' address space at every execution. Thus, in order to replace a return address in a meaningful way, an adversary must *guess* the address of the subroutine to be used. The number of possible addresses in 64-bit systems renders this guess impossible.

However, an attacker can still bypass this security mechanism by disclosing the base address of a loaded library. This can be accomplished through leaking a pointer, pointing to the library through a memory leak or buffer overread. Then, the attacker can calculate the desired function's address using the leaked address and an offset. Moreover, it has been demonstrated that even without the disclosure of an application's memory map, attacks are still feasible [22].

1.3 CFI

Those were just a few of the security oriented techniques and technologies widely used in today's systems. Though techniques like those put an end to *straight-forward* code injecting or taking control of the process, both can still be achieved with a little ingenuity and new techniques to work around those [24]. The most promising of defences seem to be the original Control-Flow Integrity (CFI) proposal by Abadi et al [11].

CFI is a principle that aims at guaranteeing that a given program will adhere to its intended execution path (Control-Flow Graph). Its techniques focus on reinforcing indirect branches, like calls and returns, so as to be immune to tampering from vulnerability exploits like buffer overflows. The ideal CFI solution, as proposed by Abadi et al. [11], can be viewed from two contexts: Forward-Edge and Backward-Edge.

Forward-Edge Control-Flow Integrity refers to the reinforcement of forward-edge branches, i.e. Indirect Call Instructions. Indirect calls utilise function pointers stored in memory to find their target. Unfortunately, these pointers are not safely stored in memory and are susceptible to corruption by tampering. Forward-Edge CFI validates that the indirect call reached a target that belongs to a group of authorised branch targets. This limits the call to extremely few indirect branch targets, hopefully none that the attacker could exploit.

Backward-Edge Control-Flow Integrity refers to the reinforcement of backward-edge branches, i.e. Return Instructions. Returns, like indirect calls, utilize a pointer stored in the stack as a branch target. This pointer is extremely vulnerable to tampering since it resides in the stack. Backward-Edge CFI keeps track of the return addresses and verifies that each return instruction did indeed reach its target.

Any attempts to implement CFI until now where either prohibitive in their performance or not secure enough as they where designed to be coarser as a way to alleviate their performance impact. Those implementations do not follow the guidelines proposed by the original CFI design and thus do not offer the level of security needed.

1.4 Contributions

This work contributes the following.

1. We present a survay of software security mechanisms implemented in hardware and argue that such mechanisms when implemented in hardware, provide enhanced security, while amortising the overhead in comparison with the software implementation.

2. We design, implement, and evaluate CFIX, a full-featured ISA for supporting processes hardened with CFI. The prototype is based on extending a SPARC SoC and it includes a hardware implementation of a shadow stack.

3. CFIX is complete and accurate. It protects both forward and backward edges, and the shadow stack implementation can handle recursion of arbitrary depth.

4. CFIX has practically negligible overhead. We evaluate CFIX with SPECInt and embeded benchmarks and we record a runtime overhead of less than 1% on average, which, to the best of our knowledge, stands for the first hardware implementation for full CFI support with low cost.

5. CFIX is policy agnostic and can deal with all idioms that usually interfere with hardening indirect jumps, such as the use of `longjmp` and `setjmp`. For the purpose of presenting CFIX in this work we enable a fine-grained CFI policy with shadow-stack support.

We propose an Ideal CFI implementation, true to Abadi's et al. original proposal. Our design is fully implemented in hardware, to add as little performance impact as possible, without sacrificing any security on both forward-edge and backward-edge control transfers. It touches lightly on the architecture of the CPU, only adding to it. Our design can only benefit from pre-existing security technologies, like ASLR, DEP, Stack Canaries, etc., and it is recommended that all those defences run alongside our own.

1.5 Organization

This work is organized as follows. In Section 2 we present software security mechanisms with hardware support, in Section 3 we discuss the generic architecture of CFIX and in Section 4 we thoroughly present the technical details for implementing the prototype. We evaluate CFIX in terms of security in Section 6 and in terms of performance in Section 5. We discuss various aspects of our current and future work in Section 8. We review related work in Section 7 and, finally, we conclude in Section 9.

Chapter 2

Background

2.1 Security mechanisms in hardware

Commodity processors include features aiming for enhanced system protection. Industry considers security as an important factor towards the success of a processing architecture. This notion is demonstrated through the increasing number of complex security mechanisms implemented in hardware over the recent years. Additionally, there are several research efforts proposing hardware security extensions, in order to tackle sophisticated threats. Practical and less intrusive research proposals usually are appealing to industries and are incorporated in commodity processors. Most of the hardware security mechanisms in the industry and the literature aim to provide hardware support for existing defensive techniques that have been implemented solely in software before. Additionally, in the literature, security proposals leverage hardware mechanisms not intended for security, proving the need for hardware assisted security.

Pushing security at the hardware domain has many benefits over its software counterparts. Hardware is immutable, thus the effort required to bypass such mechanisms is significantly higher. Almost any software security mechanism is incapable of defending the system's functionality if the operating system is vulnerable. Consequently, hardware security proposals define threat models with powerful adversaries. Taken to the extreme, the trust computing base defined by a security technique or technology can be reduced to only include a trusted processor(e.g. SGX).

Another benefit of hardware implemented security is the amortization of the runtime overhead imposed by the equivalent mechanisms in software. In many cases guaranteeing the security of a system requires thorough, computationally expensive checks. The runtime overhead renders most security mechanisms with strong guarantees impractical for deployment in real world applications, particularly in real-time critical systems. Other research efforts try to lessen the overhead by relaxing the security guarantees, however, offensive security literature proves that such approaches are defective. On the other hand, circuity dedicated to for those checks accelerates the security mechanism without sacrificing the security guarantees. Results from other hardware security implementations and our work, prove that intensive security mechanisms can be implemented, as extensions,

on existing hardware while keeping the overhead imposed within a practical margin.

2.2 Security-oriented processor features

In this section we review well established processor features, which aim to increase the security of a system. Additionally, we present new extensions, which are present in newer processors but not yet made standard in current software. Thankfully, the adoption of those technologies in upcoming software is gaining popularity. The slow progress of including those features in software, is due to the need of source code recompilation, source code refactoring and the need for the adoption of a security oriented programming styles. Furthermore, compilers have to adapt their processes to include those features in order to produce executables that are optimized in terms of performance and security and the utilization of new technologies.

2.2.1 Protection rings

CPUs support isolation between the operating system and the applications running on it. The operating system is allowed to take full control on the machines' resources and applications. On the other hand, applications run with lower privileges in order to restrain them from controlling the rest of the machine without supervision. Common RISC architectures, like ARM and SPARC, provide only two levels of isolation, which are controlled through the *supervisor* bit. The supervisor bit is set when the processor executes operating system's code and unset when the processor is occupied by an application. Several machine instructions, controlling machine specific and processor control registers, can only be executed if the supervisor bit is set. CPUs with x86 architecture include many different levels of privileges (protection rings) in order to isolate device drivers and also enable hardware assisted virtualization. The operating system always executes at ring 0 (figure 2.1).

2.2.1.1 Supervisor Mode Execute/Access Prevention (SMEP and SMAP)

In order to increase the granularity of the protection ring system present in x86 processors, in recent iterations of Intel processors, more policies have been added towards ensuring the security of a system raising the bar against potential supervisor level vulnerabilities. SMEP enforces that the operating system cannot execute user-level code, instead the CPU must switch to a higher ring level first, otherwise the CPU faults. SMAP is complementary to SMEP as it protects user-space data from being accessed from kernel code.

2.2.2 Virtual Addressing and memory segmentation

Almost every widely used system relies on virtual addressing. This constrains an application from interfering with another application's data. Different applications can reference the same `virtual` address but it will be translated to a different `physical` address according, to each application's page table. Moreover, in x86 processors it was possible

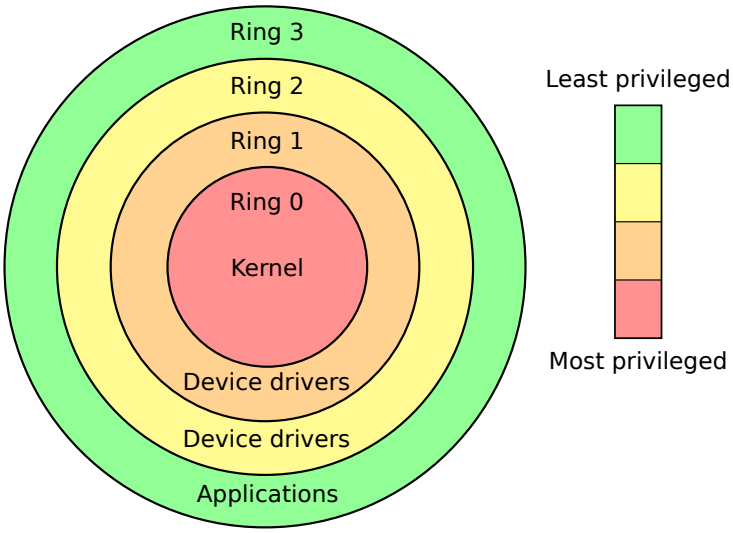


Figure 2.1: Protection rings in x86 processors By Hertzsprung at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=8950144

to isolate different parts of an application using segments. Thus, different parts of code residing within the address space of the same application cannot reference the whole application's memory layout. A typical use of this feature in applications with cryptographic operations is isolating the cryptographic keys from functions responsible for handling user input. Even if a buffer overread is possible in the code which handles the user input, due to the segmentation feature an attacker will not be able to read the cryptographic keys. In modern x86_64 Intel processors memory segmentation is considered deprecated and it is not supported anymore.

2.2.3 TrustZone

The *TrustZone* feature is available in ARM processors[25]. This feature enables the separation of execution domain to the trusted and the untrusted one. The trusted execution domain is reserved for the *trusted* system code, while third party applications are executed in the untrusted domain. This separation extends to memory and peripherals in order to achieve system wide security, e.g. a DMA capable peripheral is forbidden access to memory areas owned by trusted execution domain software.

2.2.4 Data Execution Prevention

Data Execution Prevention (DEP) is a hardware feature which halts an application if the control-flow targets data. This mechanism raised the bar for potential adversaries, since placing malicious code as data in the application's memory and then pointing the program counter to it, is not possible anymore. Thus, attackers can only rely on existing code in the application in order to achieve exploitation. Return-Oriented Programming is a sophisticated technique, widely used in modern exploits in order to bypass this mechanism.

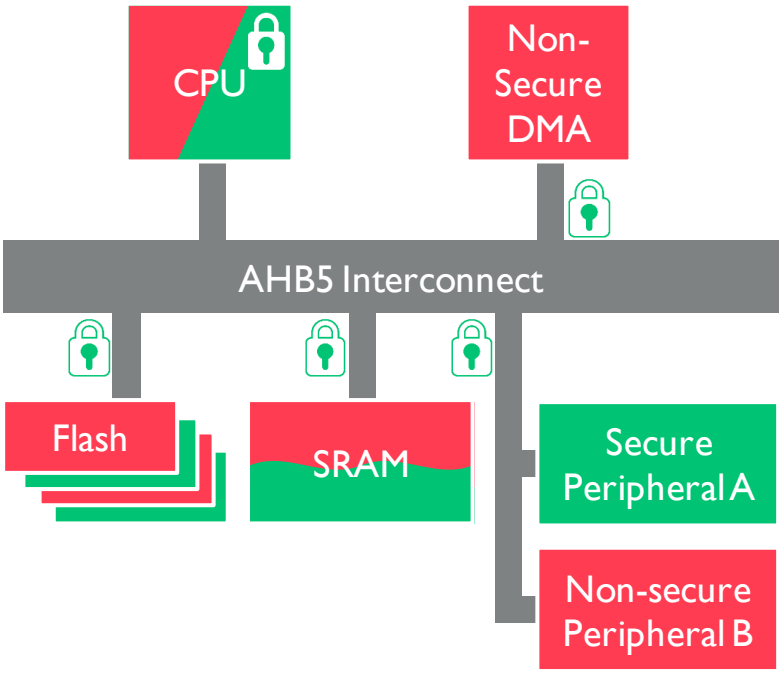


Figure 2.2: Overview of an ARM SoC with trustzone. CPU has both execution domains. Flash and SRAM define trusted and untrusted regions. Peripherals can be defined as trusted or untrusted in case of security concerns.Of course DMA should also be constrained.

DEP has different acronyms depending on the processors' brand[26]. In Intel processors it is referred to as XD bit (*eXecute Disable*). The feature is controlled through the most significant bit of a 64-bit page table entry. When this bit is set to 0, the page is assumed to hold code and can be executed. If the value is 1 the page cannot be executed since it holds data. AMD uses the same approach under the name Enhanced Virus Protection. In ARM architectures it is part of the page table entry format as XN bit (*eXecute Never*) and is placed in the page descriptor. In SPARC V8 this mechanism is used through the Reference MMU which has permission policies of Read Only, Read/Write and Read/Write/Execute in page table entries.

2.2.5 Memory Protection Extensions

Memory Protection eXtensions (MPX) is a feature introduced in Intel processors in 2015, Linux supports this feature since kernel version 3.19. Software deploying these extensions is `fortified` by associating every pointer with a lower and an upper bound. Thus, every time a pointer is dereferenced, its associated bounds are checked. If the address stored in the pointer is out of the specified bounds, a bound violation exception is raised. Since the majority of exploits depend on buffer overflows, this mechanism can effectively disclose many possible security vulnerabilities in an application.

Setting and checking the bounds is done using special MPX instructions. To deploy this feature, the application's source code must be patched with compiler intrinsics, in order to pair pointer dereferences with bound initializing and bound checking instructions. Bound checking takes as operants the pointer and a special register which holds the pointer's bounds. In order to support a large number of associated bounds, a Bounds Table residing in main memory is used. The functionality of the Bounds Table is similar to paging. The operating system stores the address of each application's Bounds Table in the Bounds Directory. When a pointer's bounds are not in one of the special bounds registers they are loaded from the bounds table. In summary, MPX extends the processor with four 128-bit bound registers, each storing a 64-bit lower bound and a 64-bit upper bound. MPX instructions are used for setting the upper and lower bounds, propagating them, moving to or from the bounds table.

The appealing security guarantees of this feature however impose a relatively high runtime overhead. If the bounds checked are not in a register, the bounds of the soon to be dereferenced pointer must be moved from the Bounds Table in one of the bounds registers. This operation is relatively expensive since the pointer address must be looked up in the Bounds Table, a process similar to a TLB miss. Additionally, bounds associated to each pointer impose significant stress to the processor's cache system. On average, MPX deployment using ICC imposed 50% overhead in SPEC benchmarks [27].

Prior to its introduction in Intel processors, bound checking was a well studied technique in the literature. Software only implementations (e.g. [28], [29]) had several drawbacks which made them not appealing for use. Non-trivial changes were required in the source code, the violation detection was limited and also the runtime overhead was very high. Hardware approaches like Hardbound [30] and CHERI's [31] processor pointer bound checking system, are based upon the concept of Fat Pointers. Herein, each pointer's value is associated with metadata, like the base address and the length, in order to verify that when the pointer changes it remains within the bounds. These hardware approaches achieved better granularity and at the same time the overhead introduced was within an acceptable margin.

2.2.6 Software Guard Extensions

Software Guard eXtensions (SGX) was proposed in 2013 and introduced in Intel's Skylake micro-architecture in 2015. Its security guarantees provide isolation between parts of a user-level application, with the further enhancement of preserving the isolation even if the drivers, operating system and BIOS are compromised. Effectively, the Trusted Computing Base is reduced to only the processor's hardware. Using this mechanism, developers can define code and data to be protected by encapsulating them in an entity called an *enclave*. SGX machine code instructions are used to create and initialize an enclave and move code and data in it. The underlying hardware provides confidentiality and integrity to the enclave. Enclave data residing in the main memory are encrypted. When moved inside the processor they are decrypted and checked for integrity. The component responsible for those operations is the Memory Encryption Engine [32]. The MEE prevents replay attacks, verifying that the data read back to the CPU from the SGX's DRAM

region are the same that were most recently written. Integrity and *freshness* are ensured using Merkle trees. Thus, even if a passive attacker snoops the bus between the processor and the main memory, the data acquired will be useless. In the case an active attacker overwrites SGX memory regions, the MEE will detect the forgery. Currently, the size allocated for enclaves is limited to 128MB. Next version of SGX (2.0) will support dynamic memory allocation. Additionally, a Seal Key, unique to each processor can be used in order to store the whole enclave when the application is terminated for future use. Another selling point for SGX is secure remote computation [33]. A user who wants to use a remote computation service, can verify that the software running in the remote enclave is legitimate and that the data sent to the remote host will not be accessible by the remote host owner. Thus, the user trusts only the author of the software running in the enclave and the processor's manufacturer (i.e. Intel). The rest of the infrastructure and the owner are untrusted.

In the literature, similar proposals exist, aiming to reduce the trust computing base to only the processor's chip. The Execute Only Memory (XOM) [34] architecture supports isolation between many containers (called compartments), by tagging every cache-line with a container identifier. Additionally, encryption and HMAC verification are integrated in the processor's memory controller, in order to verify the integrity of data residing in off-chip memory. However, XOM is vulnerable to memory replay attacks, i.e. an attacker can overwrite a memory region with old data which were previously residing in that memory region. AEGIS [35] trusts a small secure kernel in order to isolate different containers. The trusted kernel is responsible for isolating applications. Again, off-chip memory is encrypted in order to protect private data, verified, using HMAC, in order to check the integrity, and also, Merkle trees are used in order to prevent replay attacks on the off-chip memory.

2.2.7 Advanced Encryption Standard New Instruction

Secure communication amongst computer systems, relies upon strong cryptography algorithms. The most common algorithm used today is Advanced Encryption Standard (AES). AES is quite an expensive algorithm. In that notion and by the fact that all the data communicated through the internet had to be encrypted/decrypted, major semiconductor industries realized that hardware accelerated cryptography was necessary. In 2010, Intel included AES New Instructions (AES-NI) in its processors. These instructions are used in order to improve a systems throughput when processing data with AES. Other vendors like ARM and AMD also included this new instructions in their products. Although it was not an addition aiming to improve a system's security, AES-NI it boosted the fundamental cryptography algorithm used for secure communication and data protection in modern systems.

2.2.8 Last Branch Record

Last Branch Record (LBR) are register pairs on Intel processors, which implement a ring buffer in order to hold the most recent branches. Each pair holds the source and the

destination addresses of executed branches. LBR was introduced in order to assist with software debugging. However, in kbouncer [36], LBR was utilized in order to transparently detect control-flow patterns of code reuse attacks. Since LBR imposed a minimal overhead upon the protected application, this technique was very practical. Although, it was later proved to not be sufficient for defending against certain cases of ROP exploits (Section 7).

2.2.9 Dynamic Information Flow Tracking

A notable technique proposed in the literature, in order to counter buffer-overflow related exploits, is Dynamic Information Flow Tracking (DIFT). The key concept of this mechanism, is to taint memory regions where un-trusted data are residing, and track their propagation in the address space. Also, any new data resulting from computation with tainted data, also become tainted. Exploits are detected with a predefined policy, depending on the implementation of DIFT. In general, when tainted data are used in a suspicious, manner a security exception is raised. A common security exception trigger is when tainted data are used as an indirect jump operand. For example, in the case in which an attacker overwrites a return address, by exploiting a buffer overflow, input data will be tainted and the DIFT policy will detect the violation since the return address will also be tainted. In the majority of DIFT implementations, the protected application is oblivious of the mechanism, thus there is no need for source code modifications.

Of course, software implementations of such policies impose a very large overhead. In TaintCheck [37] the DIFT policy is enforced by running the protected application over Valgrind, in order to instrument Basic Blocks with code which taints memory regions during data propagation and checks for misuse of the tainted data. The runtime overhead is above a factor of ten in most benchmarks, rendering such an approach impractical. On the other hand, hardware implementation of DIFT policies can even protect complex code, such as an operating system. Meanwhile, the overhead imposed is minimal and does not affect the application's usability. In Raksha [38] the Leon3 processor is extended, in order to provide transparent protection, using DIFT, for the operating system and the user-level applications. The registers, cache and main memory are extended with a 4-bit tag in each word in order to support different DIFT policies and fine grained tracking. All ISA instructions are extended to propagate tags from input to output operands, and check tags in order to raise security exceptions when attacks are detected. Since tag propagation and checks are performed during the normal execution of the application, the overhead incurred is minimal. Negligible overhead is imposed during initialization, paging and I/O events due to the tag management. Additional overhead is introduced when deploying security exception handlers in order to suppress false positives.

2.2.10 Instruction Set Randomization

Instruction-Set Randomization (ISR) [39], [40] is a promising technique capable of mitigating code injection attacks. The main principle behind ISR is to encrypt the instructions of executables residing in the main memory and decrypting them just before execution.

This simple principle is enough to prevent the execution of arbitrary code injected to a running program by an attacker. Even if an attacker successfully injects code and diverts the control-flow to the injected code, the result will be an execution exception, since the injected code will not translate to any meaningful machine instruction.

During application loading, a random key is generated to be used when encrypting data in memory. The application keys are stored inside the process control block. The scheduler is responsible for toggling the ISR functionality bit for ISR/non-ISR applications and set the appropriate key every time an ISR application is scheduled. Finally, the page fault handler encrypts the faulting page with the application key if the owner application is ISR enabled and the page fault originates from the text segment (i.e. the faulting page contains code).

In ASIST, ISR was implemented by extending the Leon3 processor. In the processor core, the additional hardware consists of the key registers, where the scheduler loads the decryption key for the application, the ISR toggle bit, which is used to enable/disable ISR for applications with/without ISR, and hardware decryption units. Before an instruction populates the instruction cache, it is decrypted using the loaded application key. When an injection occurs, the execution of the incorrectly decoded instructions will most probably result in an illegal instruction exception which can be handled from the user-level. Additionally, the return address is encrypted during *call* instructions and decrypted it during *returns*, thereby defending against ROP attacks. Thus, if an attacker overwrites a return address, upon return the value will not decrypt correctly and the application will most probably crash.

Chapter 3

CFIX Architecture

3.1 Control Flow Integrity enforcement

Control-flow Integrity (CFI) aims at guaranteeing that the execution flow adheres to the path determined by the control-flow graph of the program. The control-flow of a program can be manipulated either on the forward-edge, when the target of an indirect jump is altered, or on the backward-edge, when a saved return address has been changed. For forward-edges we ensure that an indirect jump can target only a function entry with the appropriate label that is generated during the CFG extraction. For backward-edges we validate that the function's return instruction targets the address of the original call site. A more detailed discussion follows.

3.1.1 Forward-edge

The forward-edge is handled as discussed in the original CFI proposal [11]. Every indirectly called function is hard-coded with a label on its entry point. Before the indirect function call, the function's label is compared to a label assigned to the call site. Our approach differs in that we set the label before the indirect call executes, while the validation takes place immediately after function entry. Before the indirect control transfer, a SetPCLabel instruction, placed on the delay slot (described in subsection 4.2), stores a label in a non memory-mapped latch that resides inside the core. On the function entry, a CheckLabel instruction verifies that the label equals the one stored in the latch. If the comparison fails, a control-flow violation is detected, an exception is raised, and the system handles it appropriately.

3.1.2 Backward-edge

In the case of backward-edge control-flow integrity, we deployed a non-memory mapped stack which also resides within the core. The concept of a shadow stack is thoroughly studied in the literature [11, 15, 17]. The general concept is based on the notion that a function's return address points to the instruction lying directly below the call site. This

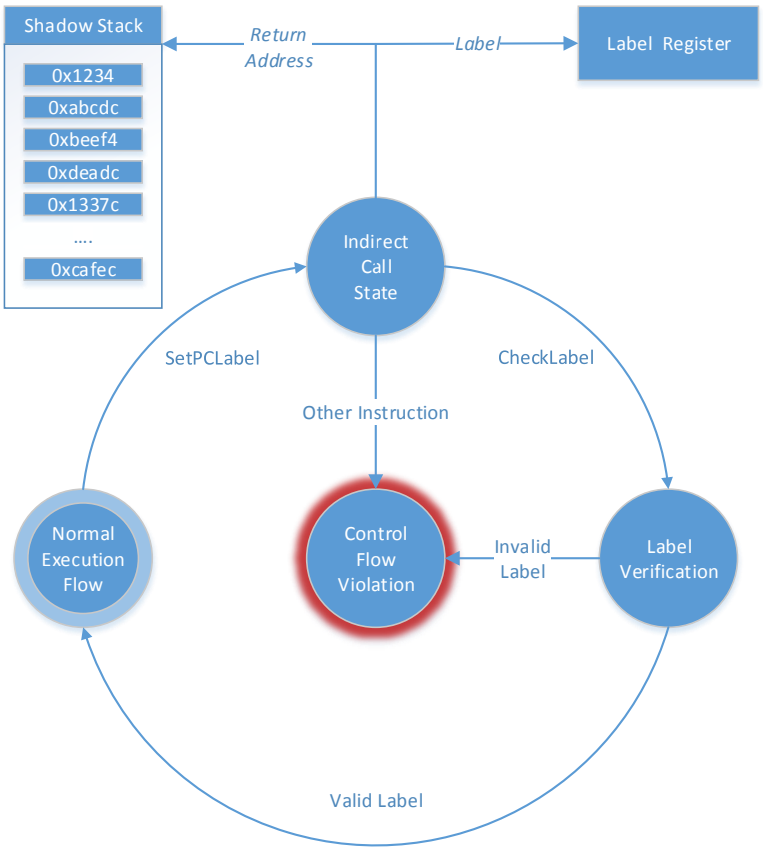


Figure 3.1: Indirect Call States. A SetPCLabel instruction is received, the appropriate memory modules are set, and the core enters a state where only CheckLabel instructions are accepted. Once a CheckLabel instruction is received, the labels are compared and execution returns to its normal flow.

is not always the case, as it is common that a function does not return to the original call site.

The shadow stack of CFIX is implemented as follows. Before a call instruction executes, a copy of the return address is pushed to the shadow stack. When the callee function returns, the return address is compared with the one on the top of the shadow stack. If they are not equal, a control-flow violation is detected and handled appropriately by the system.

Every direct call instruction is paired with a SetPC instruction placed on its delay slot. The SetPC instruction pushes the current Program Counter to the shadow stack module. After the callee function returns, a CheckPC instruction, placed in the delay slot of the return instruction, checks that the computed return address is equal with the address stored in the shadow stack, incremented by four (one instruction below the SetPC). If the check fails, a hardware exception is raised, which is handled by the supervising firmware. An alternative way to process a mismatch between the shadow stack and the main stack, is to

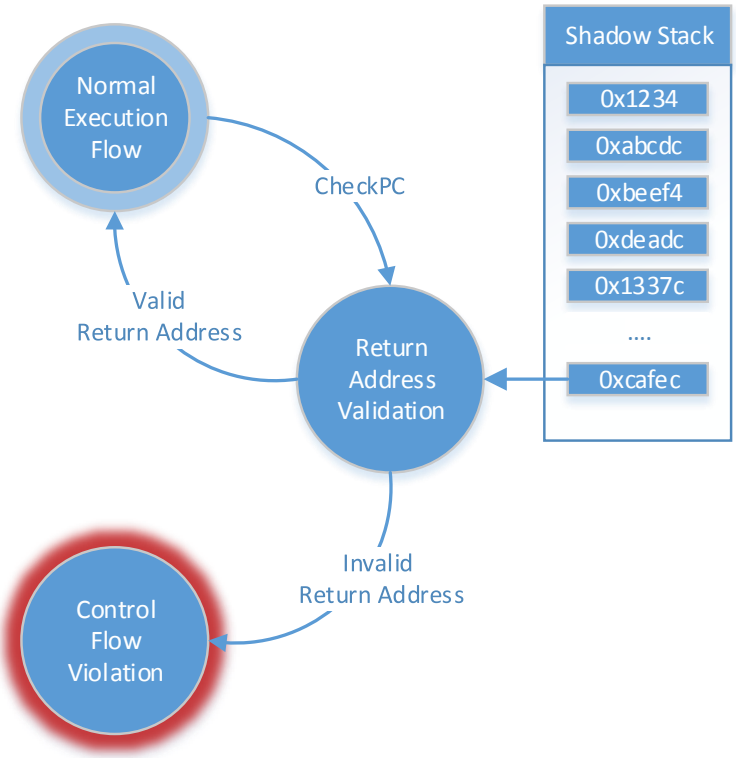


Figure 3.2: Return States. A CheckPC instruction is received, the Program Counter is compared with the top value of the stack and the execution continues normally.

silently force the address obtained from the shadow stack as the return address. Aforesaid proposition can potentially enhance our architecture with fault tolerance capabilities, since any tampering of the return address would be rectified by the hardware.

3.2 Architecture Overview

CFIX is based on a series of modifications of the Leon3 [41] core's pipeline. The architecture consists of unmapped *shadow* memory elements, more specifically a shadow stack, a shadow memory array, a shadow register, and six dedicated instructions which function upon the shadow memory blocks. The shadow stack is utilized in enforcing backward-edge CFI through the detection of control-flow changes caused by arbitrary return address modifications, e.g. buffer overflows. Likewise, a single shadow register is used for enforcing forward-edge CFI, effectively protecting the execution flow from vulnerable function pointers. The shadow memory array is used for enabling setjmp/longjmp support. To access and utilize the shadow memory blocks, we extended the SparcV8 [42] instruction set with six instructions.

3.3 ISA Extension

We extended the SparcV8 ISA with six instructions designed to provide CFI functionality to the core.

SetPC: Paired with direct call instructions. The SetPC instruction is placed in the delay slot of the call instruction it is paired with. It pushes the currently executing Program Counter (PC) to the shadow stack. If the next instruction is a CheckLabel, the SetPC instruction suppresses the CFI violation that would normally occur, since the Label Register's value has not been initialized. This functionality is useful in cases where an indirectly called function is also called directly.

SetPCLabel: Paired with indirect call instructions. This instruction is placed in the delay slot of the indirect call it is paired with. Its 18 Least Significant (LS) bits carry the label used to validate the indirect call target. As with the SetPC instruction, the current Program Counter is pushed to the shadow stack. At the same time, the 18 LS bits are stored in the Label Register to be used later for validation. If the next instruction executed is not a CheckLabel, a CFI violation occurs.

CheckLabel: Placed on the entry point of a function that is found during instrumentation to be an indirect call target. It is the only instruction that can be legally executed after a SetPCLabel. Its 18 LS bits carry the label used to validate the indirect call target. It compares the label carried on its 18 LS bits with the value stored in the Label Register. If the labels match, the register is reset and the execution continues normally, otherwise a CFI violation is detected and an exception is raised. Since the Label Register is zeroed out after every CheckLabel, and no call target is assigned zero as a label, the Label Register cannot be reused without being set again.

CheckPC: Paired with return instructions. This instruction is placed in the delay slot of the return instruction it is paired with. It compares the program counter of the next instruction executed, which will be after the branch target is reached, with the address stored in the top of the shadow stack. If PC equality is confirmed, the shadow stack is popped and the execution continues. Otherwise, a CFI violation occurs.

3.4 Shadow Stack Incompatibilities

Backward-edge control-flow integrity relies on the Call-Ret pair model of programs. That means that each time a return instruction jumps to an address different than the one it was called from, the Call-Ret pair model is violated which leads to a CFI violation. Unfortunately, there are cases that Call-Ret pairs are violated in a legitimate way.

3.4.1 Setjmp/Longjmp

The most common violation is setjmp/longjmp. When a longjmp occurs, the original stack may *unwind* by several frames. In the proposal by Davi et al. [18], a longjmp would not cause a control-flow violation but the intermediate labels would remain active,

significantly relaxing CFI. We overcome these problems by using dedicated instructions for setjmp/longjmp support, without sacrificing any security or performance.

Additionally, some designs [43] propose popping the shadow stack till a valid return address is found or it is empty, causing additional delay cycles. Furthermore, unwinding the shadow stack can lead to inconsistencies. The address required could exist more than once in the shadow stack, and since the hardware could not blindly know which of the addresses is the correct call site, it could settle on the wrong one, causing a violation later in the execution. Other proposals do not support setjmp/longjmp functionality and any such jump would be perceived as a control-flow violation.

In order to achieve *on cycle* synchronization between the shadow stack and the normal stack, we decided on the addition of two dedicated instructions and a shadow memory array. Those new instructions are paired with the call instructions to the setjmp and longjmp functions themselves.

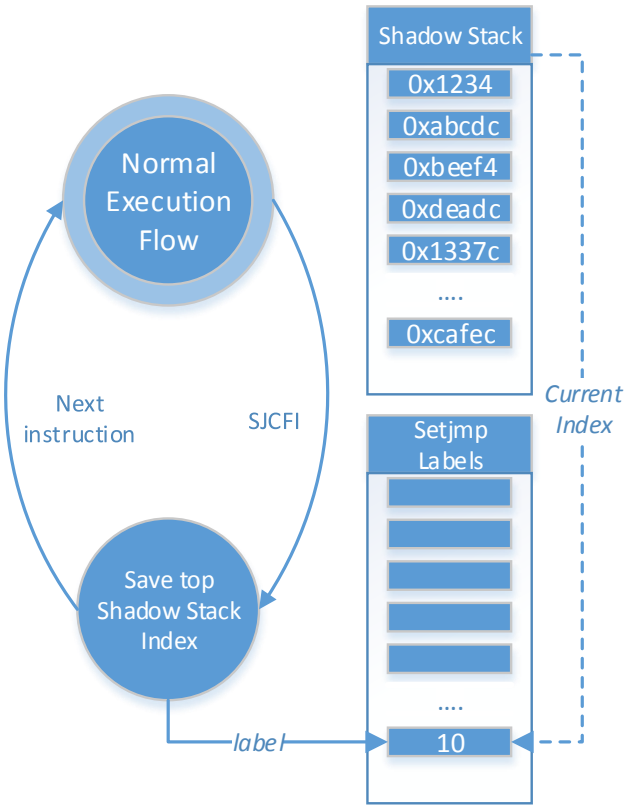


Figure 3.3: Setjmp Finite State Machine. An SJCFI instruction stores the current state of the Shadow Stack.

SJCFI: The first one, SJCFI, is paired with the setjmp function. It is placed two instructions below the call to setjmp - the instruction to which a call to setjmp would return to, when acounting for the delay slot. It carries a unique label on it 8LS bits - different from the label used for forward edge enforcement. Much like setjmp, it serves two purposes, (i)

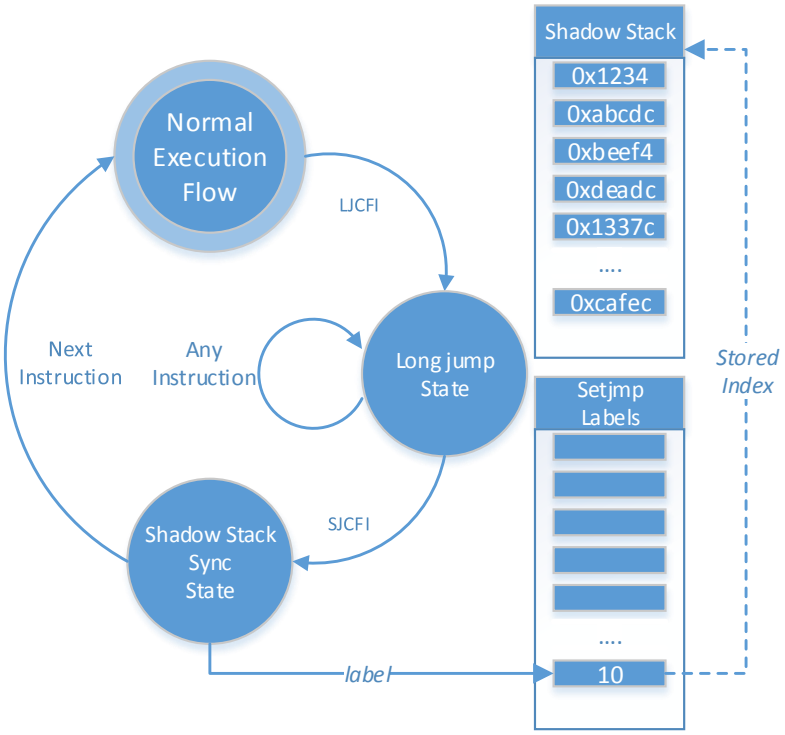


Figure 3.4: Longjmp Finite State Machine. An LJCFI instruction puts the core in a state waiting for an SJCFI instruction. The next SJCFI will not store the environment but restore it to the state that it was the last time an SJCFi instruction was executed on its own.

it sets the environment to support a longjmp, and (ii) acts as a landing point for the jump itself.

In the first case, once a setjmp returns, the first instruction executed would be SJCFI. The label is used as an index to the new memory element. During SJCFI's execution, the index of the top element of the shadow stack is stored in the new memory element using the label as an index. This pairs this particular landing point to the current state of the shadow stack. Even though the addresses remain in the shadow stack, they cannot be exploited by an attacker as the only way to use them would be to raise the index, which cannot happen without overwriting the addresses with correct ones.

SJCFI also acts as a landing point for a longjmp. Since it is placed two instructions below the call to setjmp, and a longjump will return to its equivalent setjmp call site, it will be the first instruction executed after such a jump.

For SJCFI to support long jumps, an LJCFI instruction is assumed to have been already executed. In this case, instead of reading the index of the stack and writing it to the new memory element, SJCFI reads the index from the memory element (once again using its label) and sets the stack to it. Since the index of the shadow stack corresponds with the stack frame once again, execution and CFI enforcement can continue normally. The next SJCFI instruction executed will use the first functionality unless another LJCFI was

executed before that.

LJCFI: The LJCFI instruction is only used to signify that a longjmp is underway. It is placed in the delay slot of a longjmp call and flags that a long jump is executed. After the long jump function is executed, the program counter should point to an SJCFI instruction, which will use the second of its functionalities, synchronizing the shadow stack and clearing the longjmp state flag.

The functionality of those two instructions is graphically represented in figures 3.3 and 3.4.

3.4.2 Tail-Call Elimination

Another case of Call-Ret pair violation is *tail call elimination*. As shown in figure 3.7, before calling *bar*, *foo's* return address (stored inside the o7 register) is *moved* to global register g1. When the call instruction is executed, register o7 will be overwritten with the current program counter (0x20) which serves as *bar* function's return address. Finally, in the delay slot of the call instruction, the return address of *foo* is restored in register o7. The effect of the above code snippet is that *bar* function will return to *foo* function's call site. In our design, this optimization renders the shadow-stack inconsistent with the main stack. Thus, this particular optimization has to be disabled in order to run the benchmarks. Adding support for this optimization is possible, by simply not instrumenting the eliminated call site, though that might pose a great security concern.

```
<foo>
0x0  :          ...
0x10 : mov   %o7, %g1
0x14 : call     bar
0x18 : mov   %g1, %o7
0x1C :          ...
```

Figure 3.7: SPARC V8 tail call elimination example. The calling function (foo) replaces the return address of the callee function (bar) with its own. Bar will return to the function that called foo, skipping it.

3.5 Recursion Support

The memory available to the shadow stack is finite and implemented statically inside the core, where there is no dynamic memory allocation. Hence, support for recursion could potentially be limited, since an, albeit unusual, recursion of a large depth can fill the shadow with many entries of the same address. In order to address this, we implemented an optimization that handles such cases.

Before the SetPC and SetPCLabel instructions push the current PC to the shadow stack, the stack is topped and the two addresses are compared. If the addresses are differ-

ent, the new address is pushed. Otherwise, the address is not pushed, but the current index of the shadow stack is marked as *recursive* on a separate bitmap, parallel to the shadow stack.

During the CheckPC execution, if the address currently being compared has the recursion bit activated, it is not popped from the stack. If the address comparison results in a mismatch and the top address is recursive, the top address is popped and the PC is compared with the next one. If the addresses match, execution continues normally and, if the (now) top address is not marked as recursive, it is popped. Otherwise, should the comparison again result in a mismatch, the corresponding violation is raised.

3.6 Instrumentation

Instrumentation takes place at the assembly-level of C programs. We assume that input programs are products of standard C compilers (such as, GCC and Clang), and they do not include custom assembly idioms. Instrumentation by no means is limited to C-compiler generated assembly. In fact, any assembly code is instrumentable as long as a Call-Ret-like model is sustained. The modifications where always minor and mostly consisted of moving `restore` instructions out of the delay slots, and replacing `ret` instructions with `retl` instructions, in order to account for that change. In most cases the script was able to insert our instructions in place of `nop` instructions.

In figures 3.5 and 3.6 we show the instrumentation of a direct call in the SparcV8 assembly language. The logic behind the instrumentation is fairly simple as it consists of pairing every call and return with a CFI instruction. For calls, we add a SetPC instruction below them, and for returns, a CheckPC.

We have designed our instructions to take advantage of the delay slot below branches in the SPARC V8 architecture. With that in mind, any instructions residing in the delay slot must be moved out of the slot, before the branch. The most usual case of instructions that need to be moved are *restore* instructions as they almost always reside in the delay slot of their respective return instructions.

Since the *restore* instruction changes the focus of the register window, we must compensate for it moving before the return instruction. The *ret* instruction expects to find the return address in register i7, but because the register windows have shifted, the appropriate value is now stored in register o7. Thankfully, the SPARC assembler provides another instruction with this case in mind, *retl*.

In Figures 3.8 and 3.9 we show the typical instrumentation of an indirect function call in the SparcV8 assembly language. The backward-edge components remain essentially the same, with the only modification being that the SetPC instruction is switched with a SetPCLabel instruction. But now, the forward-edge components are also in use. Specifically, SetPCLabel, storing the hard-coded label for later comparison, and the CheckLabel instruction, placed on the function entry point in order to perform said comparison.

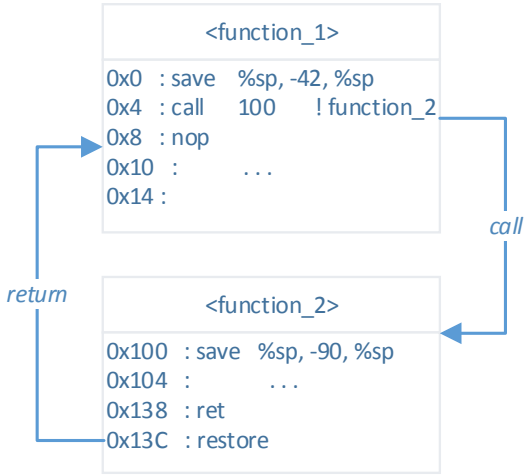


Figure 3.5: SparcV8 assembly - direct function call without CFI instrumentation.

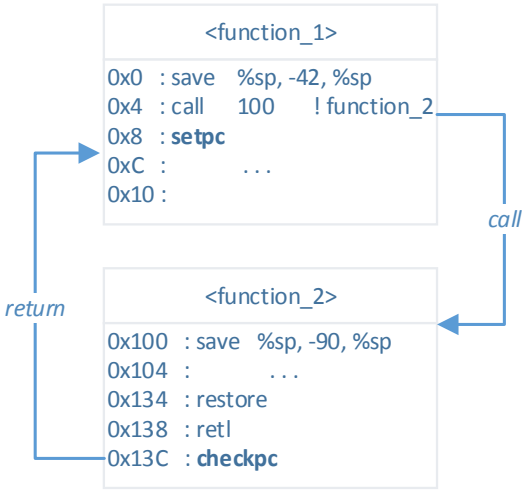


Figure 3.6: SparcV8 assembly - direct function call with CFI instrumentation. A SetPC instruction is placed on the delay slot of the call instruction, and a CheckPC on the delay slot below the return. The restore instruction is pushed above the return and the return instruction changes to account for it.

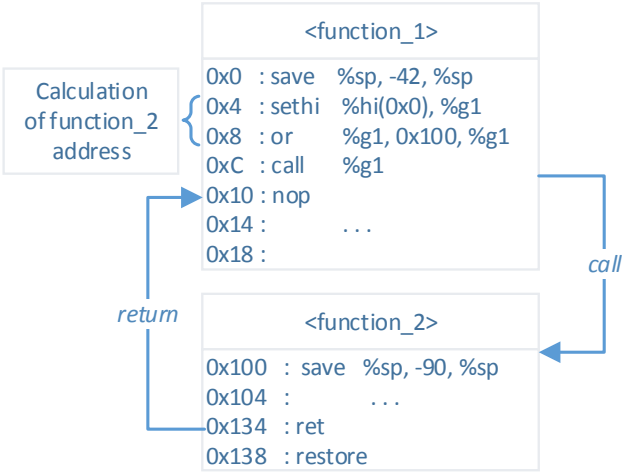


Figure 3.8: SparcV8 indirect function call without CFI instrumentation. The address in loaded in a register which is used to perform the indirect call. Otherwise, the call performs similarly to the direct call.

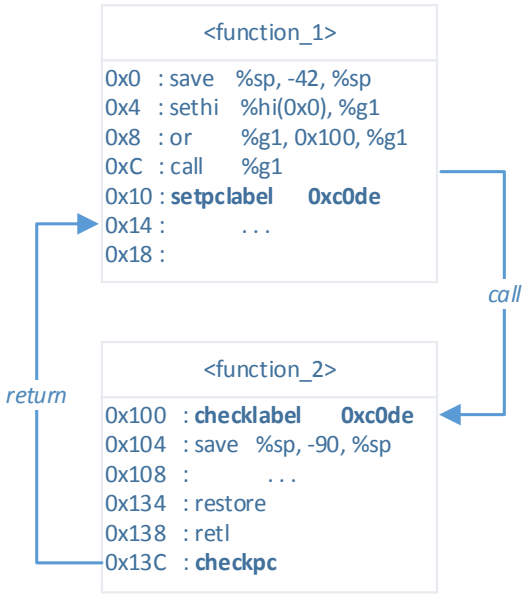


Figure 3.9: SparcV8 indirect function call with CFI instrumentation. A SetPCLabel instruction is placed on the delay slot below the indirect call. A CheckLabel instruction is placed on the entry point of the indirectly called function. Finally, a CheckPC instruction is placed in the delay slot of the return instruction.

Chapter 4

CFIX Prototype Implementation

In this section we describe the CFIX prototype implementation, we present the results of the hardware synthesis using an Virtex 6 [44] FPGA board, in terms of additional hardware needed compared to the unmodified processor, and finally we discuss how the proposed system can be easily ported to other architectures and systems.

4.1 Introduction to the Leon3 Softcore

We modified the Leon3 SPARC V8 processor [41], a 32-bit open-source synthesizable processor, to implement the security features required for a hardware-based CFI support. All hardware modifications require less than 500 lines of VHDL code. Leon3 uses a single-issue, 7-stage pipeline. Our implementation has 8 register windows, an 16 KB 2-way set associative instruction cache, and a 16 KB 4-way set associative data cache.

4.2 Delay Slot

In the SparcV8 architecture, as with many other RISC ISAs, exists the concept of a delay slot. In those architectures, any instruction directly below a branch is always executed as if before it, regardless of the result. Subsequently, the instruction slot below a branch is called a delay slot. CFIX was built with that mechanism in mind, though it is by no means a prerequisite.

4.3 Memory Element Additions

The implementation of the prototype presented in this paper requires several memory elements. Specifically, a dedicated 32 bit register, a dedicated 128*32 bit stack, a bitmap of 128 bits, and a dedicated 128*8 bit memory module. All memory elements are only accessible using the new CFI instructions of the prototype.

The register (Label Register) is used in storing the label used for indirect jump verification (forward edge). The stack (Shadow Stack) is used in storing the return addresses

of the functions currently executing, so as to add a measure of redundancy and validate return instructions (backward edge). The bitmap holds the recursion bit for the return addresses of the Shadow Stack. The third memory module is used to provide setjmp/longjmp support.

All four memory elements are not *memory-mapped*, and thus are only accessible through the use of the CFI instructions, while there is no interference with additional peripherals or supervising software. Since the memory elements do not rely on the data cache, or use any existing buses, they do not encumber the core's memory bandwidth. Also, since the elements do not reside in RAM, they can be accesed with just one cycle of delay for both reads and writes.

4.4 Leon3 Pipeline Modifications

The modifications required for supporting the new instructions, discussed in Section 3, to the core are exclusive to the pipeline. The design relies on a new `process`, the hardware equivalent of a software thread, for avoiding heavy modifications to the critical path of the pipeline. The process contains all the CFI functionality, while the Leon3 pipeline is only modified to handle the input and output for the process; such as the current and next Program Counter, signals indicating annulled instructions, exceptions, and the instructions themselves. We discuss here how each instruction is implemented.

SetPC: The basic function of the SetPC instruction is to push the current PC to the Shadow Stack during the memory stage of the execution. Additionally, during the execution stage of the pipeline, it sets a flag that is used to suppress the `Invalid Label` violation (discussed in subsection 4.5) that occurs if the next instruction executed is a CheckLabel. If the next instruction is not in fact a CheckLabel, the flag is reset. This implementation allows a function called directly to be called indirectly as well. To avoid an exception, the violation must be suppressed, as the Label Register is not currently set.

To support recursion, the instruction first tops the stack during the register access stage of the execution. If the address is the same as the current PC, it does not push it to the stack but instead marks the current index as recursive. Otherwise, it performs as previously described.

SetPCLabel: This instruction also pushes the current PC to the Shadow Stack during the memory stage and supports recursive calls as SetPC does. Additionally, SetPCLabel sets the Label Register to the value carried in its 18 LS bits. The value is extracted from the instruction during the decode stage and is set to the Label Register during the memory stage of the execution. Finally, it raises a flag that ensures that the next instruction executed is in fact a CheckLabel. If the next instruction is not a CheckLabel, then a violation is raised that will lead to an exception during the violating instruction's exception stage.

CheckPC: The CheckPC instruction serves a simple purpose. During the register access stage, it *tops* the Shadow Stack, increments the value by 4 (one instruction below the SetPC), and compares the result with the next Program Counter (nPC). If equality is confirmed, then the stack is *popped*. If the result is not the expected value, a violation is

raised leading to an exception during the exception stage.

Much like the SetPC and SetPCLabel instructions, if recursion optimization is in place, the functionality shifts. If the top address in the stack is marked as recursive, it is not popped, so that it can be used again later. If the address comparison results in a mismatch and the top address is marked as recursive, the stack is popped and another comparison is performed two cycles later, during the memory access stage. If the new comparison holds, execution continues normally and, if the top address is not recursive, it is popped. If the comparison fails again, a mismatch violation is raised during the exception stage.

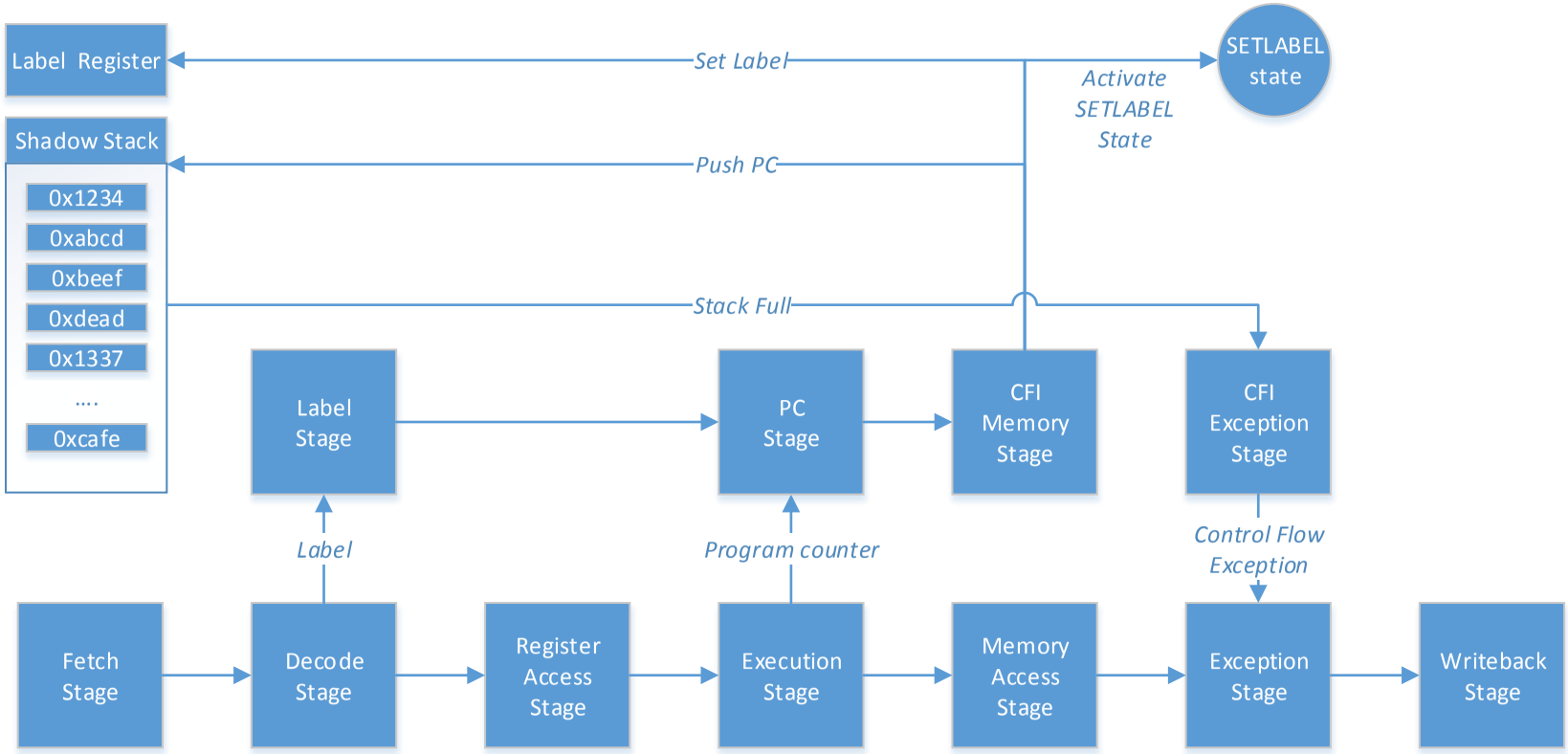
CheckLabel: This instruction, much like the SetPCLabel instruction, carries a label on its 18 LS bits. This label is extracted during the decode stage of the execution, and compared to the label stored in the Label Register during the execution stage. If label equality is not confirmed, then a violation is raised, leading to an exception. Otherwise, the Label Register is reset during the memory stage.

The CheckLabel instruction requires that a SetPC or SetPCLabel instruction was the last instruction to execute. Otherwise the Label Register is not set and its contents are zeroed. This leads to a violation, as no function is assigned zero as a label, unless a SetPC is the last instruction executed, which suppresses the violation.

LJCFI: LJCFI raises a flag to signify that a longjmp is underway. It does not carry any labels or uses any memory beyond the signal used for the flag.

SJCFI: SJCFI carries a label in its 8LS bits that is extracted at the decode stage. During the execution stage, depending on whether the flag is set by LJCFI, it either reads the top value's index from the Shadow Stack or retrieves the new index from the new memory element, using the label as an index. Finally, during the memory stage, again depending on the flag, it either stores the Shadow Stack's index to the memory element with the label as an index, or it sets the index retrieved from the new memory element to the Shadow Stack.

Figure 4.1: SetPCLabel Instruction Pipeline Modifications, without the recursion optimization in place. The label is extracted and set to the Label Register. The Program Counter is pushed to the Shadow Stack. Finally, the core enters a state were the only legal instruction that can be executed is a CheckLabel.



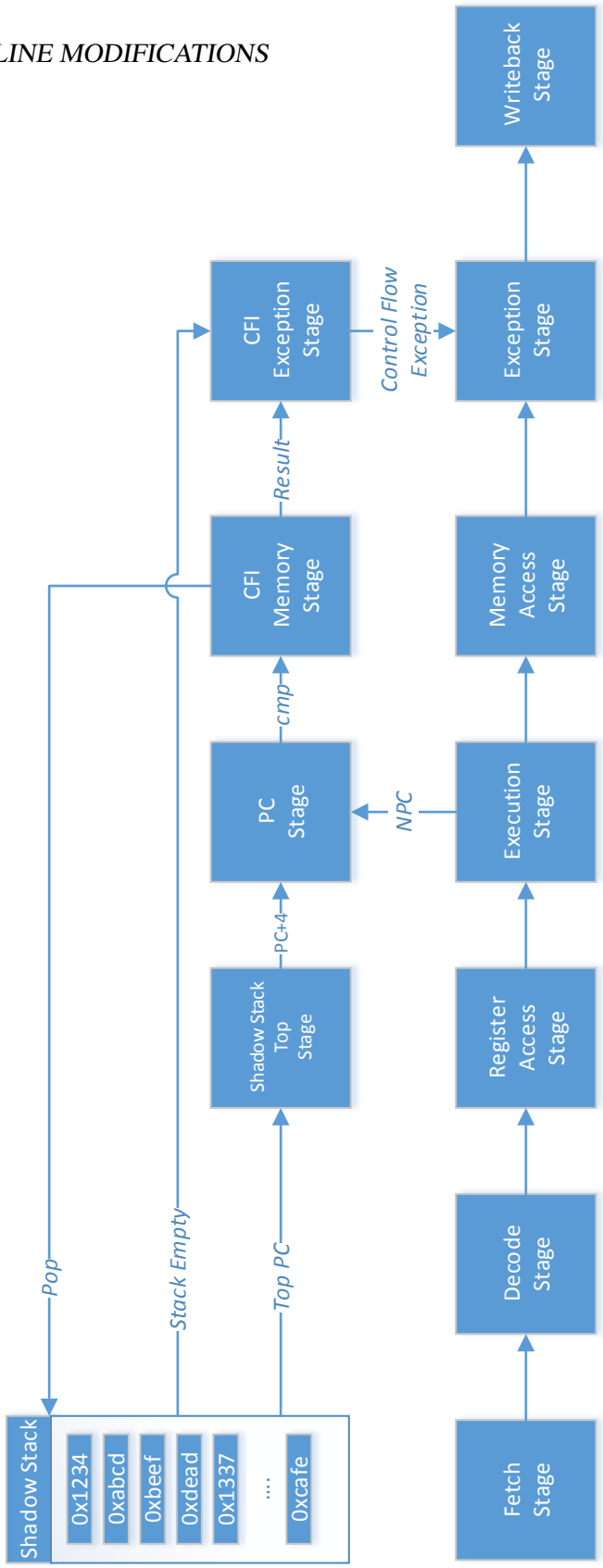
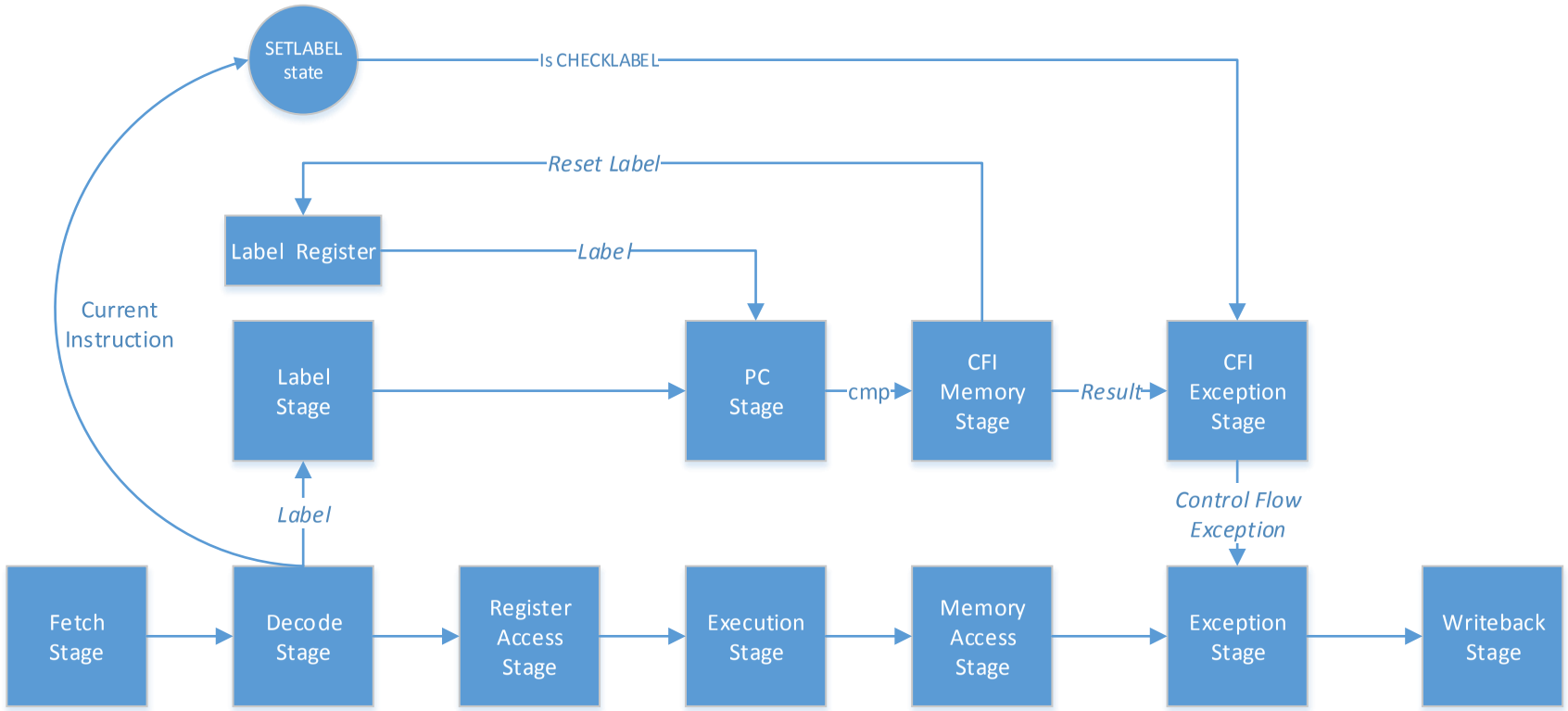


Figure 4.2: CheckPC Instruction Pipeline Modifications, without the recursion optimization in place. The Shadow Stack is topped and the value is incremented by 4. Next it's compared to the next instruction's Program Counter and finally the Shadow Stack is popped.

Figure 4.3: CheckLabel Instruction Pipeline Modifications. The core should be in a state were the only legal instruction is a CheckLabel. The label on the instruction is extracted and compared to the label stored in the Label Register. The Label Register is reset.



4.5 Violations

The various problems and errors detected during execution are summed in the following violations:

Label Mismatch: Raised when the label stored in the Label Register is not equal to the label the CheckLabel instruction carries. It can also mean that the Label Register has not been set at all. This is a forward-edge CFI violation.

PC Mismatch: Raised when a CheckPC instruction detects tampering on the return address. The address stored in the Shadow Stack is not the address to which the return instruction jumped. This is a backward-edge CFI violation.

Flow: Raised when the instruction executed after a SetPCLabel is not a CheckLabel. The indirect call targeted a function that has not been found to be a valid indirect target during instrumentation. This is a forward-edge CFI violation.

Empty: Raised when a CheckPC instruction tries to validate a return address while the stack is empty. More return addresses have been popped than have been pushed. This is a backward-edge CFI violation.

Full: Raised when a SetPC or a SetPCLabel instruction pushes a return address while the stack is full. This is not a CFI violation, but an error that is raised when the stack fills. For the implementation presented in this paper, a 128 word Shadow Stack is used and is capable to run all benchmarks. Nevertheless, a larger Shadow Stack can be easily placed in the core if needed.

In the prototype implemented on the Leon3 softcore, all violations lead to an illegal instruction exception on the exception stage of the pipeline thus, puting the Integer Unit in Error Mode and halting the execution. Alternatively, a custom exception can be easily created and handled by either the hardware or the supervising software.

4.6 Portability to Other Architectures

The design of our implementation does not actively change the core's architecture, but simply adds a few components and checks. Our implementation on Leon3 took advantage of the delay slots on branch instructions, existing in many RISC architectures, but that mechanism is by no means a prerequisite for the technology to function. The design only touches on very basic concepts of computer architecture, like the Program Counter, interrupts and exceptions, that are present in any modern core. All modifications for supporting CFIX are only additive to the processor, and rely on components present in any core. and without extending call and return instructions in order to manipulate the Shadow Stack like proposed by [43]. The logic of these instructions has been a constant since the beginning. Therefore, the design presented in this paper can be ported to any architecture with minimal effort, and, as shown in section 5.5, with a small area overhead footprint.

Chapter 5

Performance Evaluation

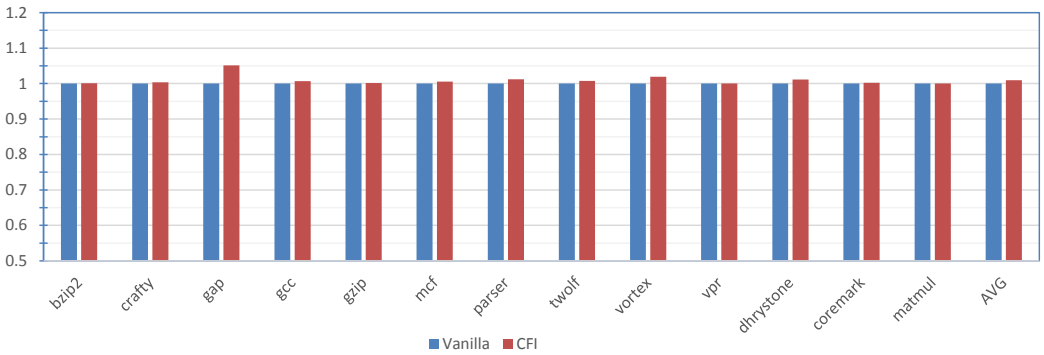


Figure 5.1: Presentation of the runtime overhead measured with our implementation compared to the runtime on a vanilla Leon3 core.

5.1 Testing Environment

We synthesized and programmed the modified Leon3 softcore on a Xilinx ml605-rev.e FPGA board. The FPGA has 1024 MB DDR3 SO-DIMM memory and the design operates at 120 MHz clock frequency. It has also several peripherals including an 100Mb Ethernet interface. Since we are targeting embedded systems, we ran all tests without an operating system present. The benchmarks are SpecInt2000 [45] and a few microprocessor-based, namely Coremark [46], Dhrystone [47], and matmul [48].

5.2 BareFS

Since the Spec suite is not designed for use in embedded systems, and running on bare-metal has the drawback of not offering the functionality of either files or command line arguments, we had to modify the code of each benchmark in order for it to be able to read its input files and arguments. The modifications included hard-coding all input files and

required command line arguments to buffers, as well as changing any instructions related to I/O so that all input comes from memory, and any possible output is either discarded, written to a new buffer, or sent to stdout/stderr.

The end purpose of the library was to simulate the functionality of file input and output by creating a pseudo file system based on strings, by filtering any function calls utilizing files, and practically acting as a proxy to stdio for any calls targeting stdin/stdout/stderr. That goal was achieved by overloading the `FILE` struct with one of our own creation that carried within it the file itself in a buffer, as well as various metadata such as file length, position, whether `EOF` (End Of File) has been reached, etc.

Likewise, every `stdio` function, that used that struct as an argument, was overloaded with our own implementations so as to handle our struct. For example, `fgetc` was overloaded to `myfgetc`. Any calls to that function where bound to use either our `FILE` struct or `stdin`. In the first case, we handled the call by returning the next character in the buffer and incrementing the position in its struct. In the second case, we made a second call to stdio and returned the value we received from there.

5.3 NOP Equivalence & Profiler Verification

Due to the architecture of the Leon3 core, our CFI instructions have the same execution time as a NOP instruction since we do not generate any main memory traffic. This allows us to perform various sanity checks during our testing phase, with regards to expected overhead. One such test consists of running all benchmarks, on an unmodified (vanilla) Leon3 core, with NOP instructions in place of our CFI instructions. All checks performed during the testing phase verified our results. Finally all results are also verified by using a profiler to count all calls, both direct and indirect, and returns executed during the benchmarks' runtimes. Again, all results were consistent.

5.4 Runtime Overhead

To measure the overall runtime overhead we ran each benchmark multiple times, instrumented with CFI instructions, on the modified core, which is programmed on the ml605-rev.e FPGA. Before each run, both the instruction and data cache were flushed. The results are depicted in Figure 5.1 and the average runtime overhead is under 1%.

We have omitted gcc and eon from SpecInt2000. In the case of gcc, CFI violations occur during normal execution, since several return addresses change after being pushed to the shadow stack. This has been confirmed by Dang et al. [15]. For evaluating gcc we count NOP instructions, since they are equivalent to CFI instructions (see Section 5.3). While the overhead reported is without the full CFI instrumentation, counting NOP instructions is really close to measuring the actual CFI instrumentation.

In the case of eon we were unable to sufficiently instrument because is was written in unstardised C++. The main problems were that we could not detect VTables and that some return addresses changed during runtime. An analysis of the code, on the assembly

level, revealed that the program loaded return addresses deirectly from memory, a few stack frames below the current one.

Interestingly, the gap benchmark came very close to reaching the maximum theoretical overhead of 6.60%. In order to measure our theoritical max runtime overhead, we measured a loop executing only indirect calls to a function which in turn immidiately executed a return instruction. The extreme overhead reported by running the gap benchmark is the result of just that - a tight loop executing a multitude of indirect calls.

5.5 Hardware Overhead

We implemented our design firstly without setjmp/longjmp support or the recursion optimization. The resulting area overhead of our implementation, as detailed by the reports of the Xilinx tools used to synthesize the design, was very low, using an additional 0.65% registers and 0.81% LUTs (look-up tables). With setjmp/longjmp support and the recursion optimization in place, the area overhead increased significantly to 2.52% registers and 2.55% LUTs. The additions to the design do not seem to add to the critical path of the processor and thus do not lower the maximum frequency that the core can achieve on the board.

5.6 Power Consumption

We measured the impact of our CFI implementation with regards to power consumption, using Xilinx XPower Analyzer tool. Our results indicate that the power consumption overhead is only 1.2%. Setjump/longjump support increases the power consumption to 1.7%.

Implementation	Power (mW)
Original Leon3	6072.11
CFI Leon3	6149.34
CFI Leon3 with SJ/LJ support	6176.92

Chapter 6

Security Evaluation

In this chapter we discuss the security guarantees provided by CFIX. In our threat model we assume that the attacker can exploit a vulnerability, either a stack or heap overflow, or use-after-free, present in the application's source code. This vulnerability can be further used to overwrite key components of the running process like return addresses, function pointers, or VTable pointers. We also consider that the attacker has successfully bypassed ASLR or fine-grained randomization [10], and has full knowledge of the process' memory layout. Nevertheless, the system enforces that (i) the `.text` segment is non-writable preventing the application's code from being overwritten, and (ii) the `.data` segment is non-executable [3] blocking the attacker from executing directly data with proper CFI annotation. Both of those principles are commonplace in today's systems preventing software exploitation. Although, code reuse attacks are designed to circumvent, and were spurred by, said defences. With that in mind, we set our goals to fill the gaps left behind by the existing defences, by securing the integrity of both forward edge and backward edge control-flow transfers.

6.1 Defence with CFI ISA extensions

By forcing every return instruction to adhere to the address stored at the top of the Shadow Stack, ROP attacks are effectively foiled. In all our tests, every change in the control flow of the application, provoked by a return instruction that was not consistent with the Shadow Stack's top value, led to a CFI violation being raised, leading to a trap in the core and the eventual termination of the execution.

Similarly, an indirect call not leading to a pre-approved function entry point would always raise a CFI violation and halt the execution. Thus, foiling again most JOP attacks by limiting the possible positions in the program that such a jump would be allowed to target. The granularity of the forward-edge protection is directly proportional to the depth of the analysis performed on compile time.

6.2 Exploitation Prevention

We run a multitude of small programs designed to violate the CFI principles in different ways, e.g. indirectly jumping with invalid labels, or no labels at all, modifying return addresses on runtime, stressing the Shadow Stack, and various others. Using behavioural simulation with Xilinx's[49] Isim tool, we had total transparency of every signal in the Leon3 softcore, and therefore the shadow memory elements themselves. We could observe every microbenchmark's effect on the Shadow Stack and the core in general. The observations were consistent with our expectations. Every control-flow violation expected was raised and detected, halting the execution. Finally, we further confirmed our observations by additonally running the microbenchmarks on the programmed FPGA board, again finding the expected results.

Chapter 7

Related Work

CFI is the base of many proposed mitigation techniques in the literature. Most of them are software-based, although there are some attempts for delivering CFI-aware processors. In this section, we discuss a representative selection of CFI solutions proposed in the literature and their limitations. A high level summary of this section is presented in table 7.1.

7.1 Coarse-Grained Approaches

Many techniques are based on relaxing the CFG a process should adhere to, thus the CFI policy enforced is coarser in comparison with the original CFI proposal [11]. Although coarse-grained CFI, as implemented in CFI for COTS binaries [13] and CCFIR [50], is practical and aims at protecting directly binaries and impose really low overhead, the security guarantees are less. Other proposals rely on hardware assistance [36, 51] in order to transparently protect applications but again, they have been proven to be weak.

7.1.1 CFI for COTS binaries

A proposal of Zhang et al. [13] is a CFI implementation that can fortify binaries by static binary rewriting. Their tool can work on commercial-off-the shelf binaries, without debug symbols and relocation information, i.e. stripped binaries. The CFI policy proposed in this work, is based on the following approach. The binary is disassembled in order to identify, all the possible addresses of indirect control flow targets. This set contains, all call preceded sites (since return addresses, target such sites), constant code pointers and computed code pointers. Every return instruction and indirect jump, is instrumented to jump to a CFI validation function, in order to ensure that the indirect control flow instruction targets one of the previously collected target addresses. Although, such policy does not ensure that return instructions will return to the original call-site. It even allows any indirect control flow instruction to target *any* address contained in the collected indirect target address set collected by the binary analysis.

7.1.2 CCFIR

Compact CFI and Randomization (CCFIR) [50] is also a CFI implementations that can protect binaries, through binary instrumentation. In CCFIR, all legal targets of indirect jumps (not returns) are collected in a *Springboard section* in a similar manner as in CFI for COTS binaries. Return instructions are only allowed to target call preceded sites, while indirect jumps are only allowed to target addresses contained in the *Springboard*. In order to hide the addresses stored in the Springboard from potential attackers, the Springboard is in a random memory area. However, memory disclosure attacks can bypass this protection. Although CCFIR has finer granularity than CFI for COTS binaries, it cannot defeat attacks utilizing call preceded gadgets [14].

7.1.3 Kbouncer

Kbouncer [36] transparently monitors applications using LBR in order to detect return instructions which target addresses not preceded by a `call` instruction. The entry of WinAPI function in instrumented in order to validate the aforementioned rule every time a system function is called. In addition, if the eight more recent branches are sort sequence of instructions followed by return (i.e. gadgets) are also perceived as attacks. This defence mechanism is able to defend only against conventional ROP attacks. Forward-edge indirect control flow branches, are not validated at all. Again, it was demonstrated that these policies are vulnerable as well [52]. Herein, an attacker can use call preceded gadgets in order to evade kBouncer's policy.

7.1.4 ROPecker

ROPecker [51], also leverages LBR in a same manner as kBouncer. Although, ROPecker, inspects the application more frequently and more thoroughly. In addition to checking at every system call, checks are performed when execution is directed towards a page which is not in the executable set of pages. ROPpecker is invoked due to the generated page fault. If an attack is not detected, the faulting page is marked as executable, the last recently used page as non-executable (in order to generate a new fault when referenced) and the execution resumes. The checks rely on detecting gadget like branching behaviour in the LBR i.e. small number of instructions followed by an indirect branch. Additionally, ROPecker tries to analyse what will happen after the execution is resumed. This is accomplished by disassembling the instruction sequence that starts at the target address, i.e. the address which caused the page fault. If an indirect branch instruction is found after a short number of instructions, the instruction stream is classified as a potential gadget. ROPecker will then emulate the instructions in order to compute the indirect branch target address and check if it also leads to a potential gadget using the aforementioned method. If it reaches an instruction sequence which is not *short*, it stops searching. Any instruction stream with less than six instruction followed by an indirect branch which does not contain any direct branches is considered gadget. ROPecker detects an attack if a sequence of 11 potential gadgets is found during a check. However, again, it was quickly demonstrated that such policy is vulnerable as well [53, 54, 55].

7.2 Selective, Fine-Grained CFI

The community realized that coarse-grained CFI does not provide sufficient security, recovering the semantics of all objects from binaries is not always possible and systems operating on binaries are all subject to more sophisticated attacks [56]. Thus focused on more fine-grained policies that can be applied at the compiler level, requiring the recompilation of the application's source code. Proposals also argued, that selectively securing only the frequently exploited elements of a running process, and not all indirect branches, as for example are VTable pointers is sufficient. However, even compiler applied fine-grained policies have been also demonstrated vulnerable, unless a shadow-stack implementation [17] is in place.

7.2.1 VTint

VTint [57] is a defense solution that protects binaries from VTable hijacking attacks. Binary rewriting is used in order to instrument security checks before virtual functions are dispatched. Binaries are analyzed in order to identify the VTables and the virtual call sites. Identified VTables are instrumented with a special ID in order to be differentiated from data and then moved to a read-only memory area, in order to be protected from arbitrary writes. Virtual call sites are instrumented with checks in order to verify that the memory area pointed by the virtual pointer is read-only and the virtual table ID is correct.

7.2.2 Virtual-Table Verification

Virtual-Table Verification [58] (VTV), is an implementation of a `relatively` fine-grained granularity forward-edge control flow integrity mechanism. It is implemented in GCC and LLVM in order to achieve better accuracy in recovering the semantics of C++ objects and emit checks at the call sites of virtual functions. To prevent attacks, VTV verifies at every virtual call the validity, of the VTable pointer being used for the virtual call, before allowing the call to execute. In order for a VTable pointer to be valid, it must point, either to the VTable for the static type of the object, or to a VTable belonging to its descendant classes. Every virtual call site, is instrumented at IR code level in order to call a verifier function before executing the call. The verifier function takes as input the VTable pointer and the set of valid VTable pointers for the site. If the VTable pointer is not in the set, the verifier function, invokes a failure function.

7.2.3 ShrinkWrap

ShrinkWrap [59], also aims to protect C++ VTables. It is built upon VTV and refines its policies in order to provide optimal protection. The main problem identified in VTV was that in cases of multiple inheritance, a call-site can have access to a large number of unrelated VTables. Thus, providing a very coarse grained protection. In order to redefine the policy of VTV and make it more fine grained, in ShrinkWrap, at every virtual call site, the set of valid legitimate virtual pointers, consists of, the VTables of the class and all the descendant *VTables* and not *all* the VTables of the descendant classes. Further, a

more optimal policy can reduce the set of legitimate VTables. The compiler *knows* the type of the object and the VTable of this object in every virtual call site. Thus, each call site can be associated with only the VTable of the class type in that call site and only the descendant VTables of that *type*.

7.3 Hardware Implementations

In this work, we do not promote a new CFI flavor, but, rather we argue that a processor architecture that supports fine-grained CFI policies *and* deploys an in-chip shadow stack can be implemented and offer strong security guarantees at a low cost (less than 1% overhead on average). Similar processors have been proposed in the literature, however none is as complete and as fast as ours.

7.3.1 Branch Regulation

In Branch Regulation [60], neither forward or backward edge control flow changes are secured adequately. Forward edges are augmented by coarse-grained CFI, which enforces that branching can target a functions entry or any point within the currently executing function. On the other hand, backward edges are protected by a shadow stack that keeps track of the program's return addresses, however, the stack itself is not secured against tampering as it resides in mapped memory. In CFIX the shadow stack is *not* mapped on the host's memory, thus it cannot be tampered.

7.3.2 HAFIX

Davi et al. [18] proposed HAFIX, a system for backward edge CFI and, unlike Branch Regulation, HAFIX does use dedicated, hidden memory elements for storing critical information. Their implementation utilizes labels to mark functions as active call sites. Labels are used as index in a bitmap, which dictates if a function is active or inactive. When a call instruction is executed the next instruction must be a `CFIBR` in order to activate the function. Only active functions are valid as return instruction targets. `CFIDEL` instructions are appended in the epilogue of each function in order to deactivate them during return. However, the aforementioned design has the disadvantage of allowing the attacker to jump to any active function. This is important, since their method also allows for an attacker, using stack unwinding, to avoid the execution of `CFIDEL` instructions in order to deactivate functions and eventually mark every function as an active one, thus effectively permitting jumps anywhere in the program, and therefore being possibly vulnerable. For forward-edge control flow transfers they rely upon VTV [58]

7.3.3 XFI

Budiu et al. [61] propose the usage of hard-coded labels for both forward edge and backward edge control-flow integrity enforcement. The usage of labels for backward-edge protection certainly limits the attacker, but still allows him to take advantage of functions

that are called by many call sites, such as `memcpy`. Essentially, this implementation is vulnerable, since it lacks of a shadow-stack implementation.

7.3.4 NSA's CFI

NSA's proposal on hardware CFI [62] facilitates a shadow stack to protect return addresses and landing point instructions to augment indirect-call transfers. In order to improve the flexibility of the shadow stack, they propose the use of a shadow MMU that will handle the management of the shadow memory. However, their proposal lacks granularity on forward-edge flow integrity, thus an attacker can point an indirect branch on any landing point instruction.

7.3.5 Intel CET

Finally, in June 2016, Intel announced Control-flow Enforcement Technology [63] (CET). In CET a shadow stack is defined in order to protect backward-edge control flow transfers in a manner similar to our design. In contrary to our design, when CET is enabled, `call` instructions are responsible for pushing the return address in the shadow stack as well as in the original stack. `Ret` instructions pop the shadow stack and ensure that it matches the return address acquired from the application's stack. In case of mismatch an exception is raised and the execution of the application stops. The shadow stack's integrity is protected by the MMU in order to prevent an adversary from overwriting the return addresses residing in it. Any memory instruction, trying to access the contents of the shadow stack is blocked by the MMU and a page fault is raised. In order to protect forward-edge control flow transfers `ENDBRANCH` instruction is used to mark the legitimate landing points for call and indirect jump instructions within the applications code. When a jump is issued CET enters `WAIT_FOR_ENDBRANCH` state. If an `ENDBRANCH` instruction is not the next instruction in the program stream, the processor raises a control protection fault.

Proposal	SW/HW	Forward Edge	Backward Edge	Setjmp Longjmp Support	Source code recompilation	Shadow Stack
Abadi et al. CFI	SW	FINE	FINE	N	Y	Y
CFI for COTS Binaries	SW	COARSE	COARSE	Y	N	N
CCFIR	SW	COARSE	COARSE	Y	N	N
Kbouncer	HW assisted	COARSE	COARSE	Y	N	N
ROPecker	HW assisted	COARSE	COARSE	Y	N	N
VTint	SW	COARSE	N	Y	N	N
VTV	SW	COARSE	N	Y	N	N
ShrinkWrap	SW	FINE	N	Y	Y	N
Branch Regulation	HW	COARSE	FINE	N	Y	Y
HAFIX	HW	N	COARSE	Y	Y	N
XFI	HW	FINE	COARSE	Y	Y	N
NSA's CFI	HW	COARSE	FINE	NA	Y	Y
Intel CET	HW	COARSE	FINE	NA	Y	Y
CFIX	HW	FINE	FINE	Y	N	Y

Table 7.1: A summary of CFI related proposals.

Chapter 8

Discussion & Future Work

CFIX's design does not offer support for multi-threaded environments. A single shadow stack located in the core is not sufficient to store the return addresses for all the processes that share the processor. Implementing an array of shadow stacks inside the core would be a step towards achieving this functionality. Unfortunately, such a hardware implementation is not feasible, due to the substantial area overhead it would introduce; the array would have to be large enough to store the shadow stack for every active process.

This approach can be easily implemented by having a small array of shadow stacks indexed by the processes' IDs. When a context switch occurs, the operating system has to store the new running process' ID to a memory-mapped register that is visible to the control-flow integrity pipeline. The pipeline would, in turn, use it to select the appropriate shadow stack for the running process. When the process is terminated, a *cfiexit* instruction will be issued in order to invalidate the process' slot in the shadow stack array. Additionally, several software stacks could be integrated in one hardware stack. Instead of using an array of shadow stacks, the operating system could use one shadow stack for storing return addresses along with their respective process ID. Once a process terminates, all stale records contained at the shadow stack should be cleaned up by the operating system. Enabling multiple shadow stacks is part of our future work.

The most novel approach, that handles multi-threaded environments, is proposed by IAD [62]. They conclude that the optimal design for a shadow memory assisted architecture has to derive a large subset of the logic from the existing MMU subsystem. In essence, the core is augmented with a second MMU(Shadow MMU), that reserves a small part of the system's memory, and denies access to it for everything but itself and the the set of CFI instructions from the core's extended ISA.

Compared to our design, the above approach would provide the same level of security, but would trade-off performance for multi-threading support. The performance degradation is owed to the fact that the CFI instructions would now operate on the *slow* system memory, and not the initial memory located inside the processor's core. An obvious optimization, is to include enough memory for one process inside the core, much like in our current design, and utilize the shadow MMU in order to swap the CFI values of the executing thread at every context switch. With this optimization, the shadow MMU ar-

chitecture would add minor performance overhead over our initial design, considering the overhead imposed by the context switch itself, and therefore it could be essentially used on top of CFIX. We plan to explore this integration in our future work.

Chapter 9

Conclusion

In recent years, a trend has begun among the leading hardware manufacturers, in developing and providing hardware-assisted security enforcement techniques in commodity hardware; in particular, processors. With the advent of the Internet Of Things and with our planet and lives becoming ever more interconnected, this is a very welcome trend indeed.

In this work we aimed to provide security solutions against one of the most timeless, yet still prevalent software exploits found in the wild; Code-Reuse attacks through Stack-Smashing. Proposed software-only defenses, either lack in security assurances, or impose prohibitive runtime overheads, making them impractical for deployment in any system. Not wanting to propose another impractical security scheme, we opted to follow the trend set by the Industry and pushed a well studied security technique to the hardware domain.

We began by presenting a small rundown of the history of stack smashing attacks and the existing defenses against them. We explained the concept of Control Flow Integrity and its strong security guarantees, but, also, its shortcomings, with regards to execution time overheads, that render it a very poor choice for enhancing a system's security. We discussed the advantages of using hardware enforced principles, to gain the benefits of the lower runtime overheads imposed during execution, as well as the extra security provided by the immutability of hardware. Finally, before presenting our own work, we discussed the state-of-the-art in hardware security, and how those techniques, while substantially raising the bar for any attacker, can not fully protect a system, even when used in conjunction with one another.

Control Flow Integrity is a principle that was designed to offer complete protection against any sort of control-flow hijacking attacks, that hinges on the malicious manipulation of control-flow variables by an adversary. Indeed, CFI can offer a perfect security scheme for protecting against such attacks, as long as its principles are followed to the letter, and not relaxed so as to amortize its, decidedly, large runtime overhead. Using what we previously discussed, namely that hardware-assisted security techniques can greatly alleviate such overheads, we set forth to outline a CFI implementation that would make no compromises to its security assurances, even strengthening them further, while maintaining a very manageable performance degradation. We designed our proposed CFI scheme

to also make use of the immutability of hardware, requiring that all memory elements needed, be unmapped; hence, invisible to the software, and by extension, any potential attackers.

To achieve these goals, we extended the SPARCv8 ISA of a Leon3 processor, with new instructions and memory elements, to provide the functionality envisioned. We overcame the hurdles and edge-cases that arose without lowering the level of security offered. The implemented design successfully met the requirements we had set for it, offering perfect Control Flow Integrity, while keeping the average runtime overhead just under 1%.

Bibliography

[1] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "Hcfi: Hardware-enforced control-flow integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16. New York, NY, USA: ACM, 2016, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2857705.2857722

[2] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, p. 365, 1996.

[3] S. Andersen and V. Abella, "Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, Data Execution Prevention," Microsoft TechNet Library, September 2004, http://technet.microsoft.com/en-us/library/bb457155.aspx.

[4] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[5] 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*. ACM, 2011, pp. 30–40.

[6] PaX Team, "Address Space Layout Randomization (ASLR)," 2003, http://pax.grsecurity.net/docs/aslr.txt.

[7] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 601–615. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.41

[8] 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*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382216

[9] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 571–585. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.39

[10] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, 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*, May 2013.

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[12] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The devil is in the constants: Bypassing defenses in browser jit engines." in *NDSS*. The Internet Society, 2015. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2015.html#AthanasakisAPPI15

[13] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries." in *Usenix Security*, 2013, pp. 337–352.

[14] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 575–589.

[15] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *ACM Symposium on Information, Computer and Communications Security, ASIACCS*, vol. 15, 2015.

[16] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 901–913. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813646

[17] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 161–176. [Online]. Available: http://blogs.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[18] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: hardware-assisted flow integrity extension," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 74.

[19] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *Usenix Security*, vol. 98, 1998, pp. 63–78.

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

[21] 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*. ACM, 2008, pp. 27–38.

[22] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 227–242.

[23] P. Team, "Pax address space layout randomization (aslr)," 2003.

[24] 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*. ACM, 2004, pp. 298–307.

[25] "ARM trustzone," https://www.arm.com/products/security-on-arm/trustzone.

[26] "No Execute bit," https://en.wikipedia.org/wiki/NX_bit.

[27] "Performance Evaluation of MPX," https://intel-mpx.github.io/performance/.

[28] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 290–301. [Online]. Available: http://doi.acm.org/10.1145/178243.178446

[29] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, *Dependent Types for Low-Level Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 520–535. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-71316-6_35

[30] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the c programming language," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 103–114. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346295

[31] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 457–468.

[32] S. Gueron, "A memory encryption engine suitable for general purpose processors." *IACR Cryptology ePrint Archive*, vol. 2016, p. 204, 2016.

[33] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, 2013.

[34] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 178–192, 2003.

[35] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *Information Security Technical Report*, vol. 10, no. 2, pp. 63–73, 2005.

[36] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 447–462. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas

[37] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[38] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 482–493.

[39] 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*. ACM, 2003, pp. 272–280.

[40] 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*. ACM, 2013, pp. 981–992.

[41] Gaisler Research, "Leon3 synthesizable processor," http://www.gaisler.com.

[42] "The SPARC Architecture Manual, Version 8," www.sparc.com/standards/V8.pdf.

[43] H. Özdoganoglu, T. Vijaykumar, C. E. Brodley, B. Kuperman, A. Jalote *et al.*, "Smashguard: A hardware solution to prevent security attacks on the function return address," *Computers, IEEE Transactions on*, vol. 55, no. 10, pp. 1271–1285, 2006.

[44] Xilinx, "Xilinx Virtex 6 ml605 rev-e Evaluation Board," http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf, 2012.

[45] Standard Performance Evaluation Corporation (SPEC), "SPEC CINT2000 Benchmarks," http://www.spec.org/cpu2000/CINT2000.

[46] EEMBC, "Coremark Benchmark," https://www.eembc.org/coremark/.

[47] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.

[48] J. Burkardt, P. Puglielli, and P. S. Center, "Matmul: An interactive matrix multiplication benchmark," *degrees from BITS, Pilani. He is a Fellow of the Institution of Engineers (India), Fellow of National Academy of Engineering (FNAE), Fellow of National Academy of Sciences (FNASc), Life Member ISTE(LMISTE). Professor Kothari has published/presented*, vol. 640, 1995.

[49] Xilinx, "ISE Simulator (ISim)," http://www.xilinx.com/tools/isim.htm.

[50] 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 *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.

[51] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against ROP attacks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*, 2014. [Online]. Available: http://www.internetsociety.org/doc/ropecker-generic-and-practical-approach-defending-against-rop-attacks

[52] E. Göktaş, 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 *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 417–432. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/goktas

[53] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 401–416. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi

[54] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 385–399. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini

[55] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, "Evaluating the effectiveness of current anti-rop defenses," in *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, 2014, pp. 88–108. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11379-1_5

[56] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[57] Chao Zhang, Chengyu Songz, Kevin Zhijie Chen, Zhaofeng Cheny, and Dawn Song, "Vtint: Protecting virtual function tables' integrity," in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[58] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc and llvm," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 941–955. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671285

[59] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos, "Shrinkwrap: Vtable protection without loose ends." in *ACSAC*. ACM, 2015, pp. 341–350. [Online]. Available: http://dblp.uni-trier.de/db/conf/acsac/acsac2015.html#HallerGAPB15

[60] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 2012, pp. 94–105.

[61] M. Budiu, U. Erlingsson, and M. Abadi, "Architectural support for software-based protection," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 2006, pp. 42–51.

[62] "Hardware Control Flow Integrity for an IT Ecosystem," https://github.com/iadgov/Control-Flow-Integrity/tree/master/paper, 2015.

[63] "Control-flow Enforcement Technology Preview," https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2016.