UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCES AND ENGINEERING

Hardware-Assisted Security Mechanisms for Memory Vulnerabilities

by

George Christou

PhD Dissertation

Presented

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Heraklion, July 2023

UNIVERSITY OF CRETE

DEPARTMENT OF COMPUTER SCIENCE

Hardware-Assisted Security Mechanisms for Memory Vulnerabilities

PhD Dissertation Presented

by **George Christou**

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

APPROVED BY:

Author: Georgios Christou

Supervisor: Sotiris Ioannidis, Associate Professor, Technical University of Crete

---

Committee Member: Evangelos Markatos, Professor, University of Crete

---

Committee Member: Vassilis D. Papaefstathiou, Assistant Professor, University of Crete

---

Committee Member: Angelos Billas, Professor, University of Crete

---

Committee Member: Georgios Vasiliadis, Assistant Professor, Hellenic Mediterranean University

---

Committee Member: Polyvios Pratikakis, Assistant Professor, University of Crete

---

Committee Member: Nikos Vasilakis, Assistant Professor, Brown University

---

Department Chairman: Antonis Argyros, Professor, University of Crete

Heraklion, July 2023

Acknowledgments

I would like to thank my supervisor, Professor Sotiris Ioannidis for his immeasurable help and guidance over the past decade. I also feel grateful to Professor Evangelos P. Markatos for his support during my studies. I am also indebted to the remaining members of my committee, Vassilis D. Papaefstathiou, Angelos Billas, Georgios Vasiliadis, Polyvios Pratikakis and Nikos Vasilakis for their invaluable feedback and comments during my defense. I also want to thank Vassilis Kemerlis, Nick Kossifidis and Elias Athanasopoulos for the successful collaboration on publications included in this thesis.

I also want to thank past and present members of Distributed Computing Systems in Foundation for Research and Technology Hellas (FORTH) and the Microprocessor and Hardware Laboratory in Technical University of Crete for their support all these years. Equivalently, I also feel grateful to the Computer Architecture and VLSI Systems (CARV) laboratory members for providing equipment and assistance on many projects related to this thesis and making me feel like a member of their laboratory.

Additionally, I want to thank the members of the RISC-V foundation, for their support on my effort to include my research in the RISC-V architectural standards. At this point, I want to convey my deepest regards to Vedvyas Shanbhogue for sharing his profound knowledge on computer architecture during our joint work in the Shadow Stack and Landing Pads Task Group.

Finally, I would like to thank my family and friends for their deep support.

Abstract

Security is essential in today's computing systems. Nearly every aspect of modern life is associated with computing devices. This trend is not expected to slow down in the near future, conversely, it is expected to continue expanding. Thus, it is important to ensure that modern systems are secure against cyber-threats, especially considering that computing devices are responsible even for life-critical tasks (e.g., medical devices, smart cars, *etc.*).

Security relies on checking various conditions during an application's execution as well as various computations on the application's memory (e.g., hashing, cryptography, *etc.*). A plethora of effective security mechanisms has been designed and implemented solely in software. However, the additional security checks and operations are not cheap and cause runtime performance overhead as well as increased power consumption. In an effort to reduce the imposed overheads relaxed and lightweight s ecurity variations of these strategies have been proposed, but they often prove ineffective and easy to bypass with new, more sophisticated exploitation techniques. Researchers and industry providers strive to find the golden ratio between security and overall functionality of the system. This is not an easy task and it has been long proven that effective strategies relying only on software, often fail to achieve both of these goals. Modern CPUs introduce progressively more architectural extensions which aim to accelerate certain *heavy* operations. Thus, one could argue that pushing parts of security mechanisms in the hardware domain is a promising approach, in order to offer strong security guarantees with minimal runtime overhead.

In this dissertation, we explore the design of hardware assisted security mechanisms in order to protect systems against common exploitation techniques. Our work can be divided in two categories of mechanisms. First, we utilize architectural extensions already present in commodity off the self hardware, even if they were not originally designed for security purposes. Second, we design and implement our own hardware extensions that aim to enhance the performance of promising security strategies which were originally implemented solely in software. The techniques we explored prevent memory related vulnerabilities from escalating to successful exploitation of the system. In summary, we present a lightweight main memory encryption mechanism that leverages widely available cryptographic accelerators and MMU components in order to prevent attackers with physical access from disclosing sensitive data. We then explore intra-process isolation through leveraging hardware assisted user-level memory partition in order to preserve memory safety in managed runtime environments when libraries written in non memory safe (or type safe) languages are loaded. Furthermore, we design and implement cryptographi-

cally resistant architecturally assisted Instruction Set Randomization in to prevent Code Injection and Code Reuse attacks. Finally, we design and implement a complete, policy agnostic Control Flow Graph based Control Flow Integrity instruction set and we discuss how we adapted our work in order to form the specification for CFI in RISC-V architecture. The evaluation of our work and the tendency of CPU providers to include architectural extensions for security verifies that our approach is promising for defending against the ever-expanding threat landscape.

Keywords: Architectural Extensions, Systems Security

Supervisor: Sotiris Ioannidis
Associate Professor
School of Electrical and Computer Engineering
Technical University of Crete

Περίληψη

Η ασφάλεια είναι πολύ σημαντική στα σημερινά υπολογιστικά συστήματα. Σχεδόν κάθε πτυχή της καθημερινότητας συνοδεύεται με τη χρήση κάποιας υπολογιστικής συσκευής. Αυτή η τάση δεν φαίνεται ότι θα σταματήσει σύντομα, αντιθέτως αναμένεται να ενισχυθεί. Επομένως είναι σημαντικό τα υπολογιστικά συστήματα να είναι ασφαλή ενάντια σε κυβερνοεπιθέσεις, ειδικά αν σκεφτεί κανείς ότι χρησιμοποιούνται ακόμα και για ζωτικής σημασίας εφαρμογές (π.χ. ιατρικές συσκευές, έξυπνα αυτοκίνητα, κ.τ.λ.).

Η ασφάλεια συνήθως αποτελείται από διάφορους ελέγχους κατά τη διάρκεια εκτέλεσης μιας εφαρμογής καθώς και διάφορους υπολογισμούς στην μνήμη της εφαρμογής (π.χ. κατακερματισμός, κρυπτογραφία, και άλλα) Αρκετοί αποτελεσματικοί μηχανισμοί έχουν σχεδιαστεί και υλοποιηθεί πλήρως σε λογισμικό. Όμως, οι επιπλέον έλεγχοι και υπολογισμοί για την ασφάλεια δεν είναι φτηνοί υπολογιστικά και επιβραδύνουν σημαντικά το σύστημα και υπολογιστικά αλλά και από άποψη κατανάλωσης ενέργειας. Σε μια προσπάθεια να μειωθεί η επιβάρυνση του συστήματος, έχουν προταθεί μηχανισμοί με λιγότερο εντατικούς ελέγχους, όμως συνήθως αποδεικνύονται ανεπαρκείς και εύκολα παραβιάσιμοι μέσω νέων εξελιγμένων επιθέσεων. Οι ερευνητές και η βιομηχανία πασχίζουν να βρουν την χρυσή τομή μεταξύ ασφάλειας και λειτουργικότητας του συστήματος. Αυτό δεν είναι εύκολο και έχει αποδειχθεί ότι στρατηγικές που βασίζονται αποκλειστικά σε λογισμικό αποτυγχάνουν και στους δυο αυτούς στόχους. Οι σημερινοί επεξεργαστές περιλαμβάνουν συνεχώς νέες τεχνολογίες που σκοπό έχουν να επιταχύνουν διάφορες απαιτητικές διεργασίες. Επομένως, κάποιος θα μπορούσε να πει ότι το να υλοποιηθούν τμήματα ενός μηχανισμού ασφαλείας σε υλικό, θα μπορούσε να προσφέρει υψηλού επιπέδου ασφάλεια ενώ παράλληλα η επιβάρυνση του συστήματος θα είναι ελάχιστη.

Σε αυτή τη διατριβή, εξερευνούμε τον σχεδιασμό αρχιτεκτονικά υποβοηθούμενων μηχανισμών ασφαλείας προκειμένου να προστατεύσουμε ένα σύστημα από συνήθεις τεχνικές παραβίασης. Η εργασία μας μπορεί να διαχωριστεί σε δυο κατηγορίες μηχανισμών. Στην πρώτη κατηγορία, αξιοποιούμε για ασφάλεια, αρχιτεκτονικές επεκτάσεις που είναι ήδη παρούσες σε επεξεργαστές του εμπορίου, ακόμα και αν ο αρχικός σκοπός τους ήταν διαφορετικός. Στη δεύτερη κατηγορία, σχεδιάζουμε και υλοποιούμε δικές μας αρχιτεκτονικές επεκτάσεις έτσι ώστε να επιταχύνουμε υποσχόμενες στρατηγικές ασφαλείας που έχουν σχεδιαστεί αρχικά μόνο με λογισμικό. Οι τεχνικές που μελετήσαμε προλαμβάνουν την επιτυχή αξιοποίηση ευπαθειών μνήμης από επιτιθέμενους. Περιληπτικά, παρουσιάζουμε ένα μηχανισμό κρυπτογράφησης κύριας μνήμης όπου αξιοποιούμε κοινές αρχιτεκτονικές επεκτάσεις με σκοπό να αποτρέψουμε την κλοπή δεδομένων από επιτιθέμενους με φυσική πρόσβαση στο σύστημα.Έπειτα, μελετήσαμε την απομόνωση τμημάτων κώδικα μιας διεργασίας μέσω ενός αξιοποίησης αρχιτεκτονικά υποστηριζόμενου μηχανισμού διαχείρισης

μνήμης σε επίπεδο χρήστη, προκειμένου να διατηρήσουμε την ασφάλεια μνήμης σε περιβάλλοντα εκτέλεσης με διαχείριση, όταν φορτώνουν βιβλιοθήκες γραμμένες σε γλώσσες χαμηλού επιπέδου. Επιπροσθέτως, σχεδιάσαμε και υλοποιήσαμε ένα κρυπτογραφικά ασφαλές αρχιτεκτονικά υποστηριζόμενο μηχανισμό τυχαιοποίησης συνόλου εντολών προκειμένου να προστατεύσουμε εφαρμογές από επιθέσεις εισαγωγής κώδικα και επαναχρησιμοποίησης κώδικα Τέλος, σχεδιάσαμε και υλοποιήσαμε μια επέκταση συνόλου εντολών για τον έλεγχο ακεραιότητας ροής εκτέλεσης, βασισμένο σε γράφους ροής εκτέλεσης, παράλληλα μελετάμε την εφαρμογή αυτής της δουλειάς στα πλαίσια των προδιαγραφών της αρχιτεκτονικής RISC-V. Η αξιολόγηση της δουλειάς μας καθώς και η τάση των εταιρειών επεξεργαστών να συμπεριλαμβάνουν αρχιτεκτονικές επεκτάσεις για ασφάλεια, επαληθεύει ότι η πρακτική μας είναι πολλά υποσχόμενη στο να προστατεύει συστήματα από το συνεχώς αυξανόμενο πεδίο απειλών.

Λέξεις Κλειδιά: Αρχιτεκτονικές Επεκτάσεις, Ασφάλεια Συστημάτων

Επόπτης: Σωτήρης Ιωαννίδης
Αναπληρωτής Καθηγητής
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Πολυτεχνείο Κρήτης

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 cyber systems 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. Even without exploiting software, adversaries can launch powerful attacks and disclose sensitive information e.g. passwords, session cookies, etc. residing in the memory. Moreover, the expansion of edge and cloud computing technologies has significantly broadened the trusted computing base of a cyber infrastructure and consequently the attack surface.

More than a decade ago, exploiting software was as easy as just simply smashing the stack [141]. An attacker could fill a vulnerable buffer located in the stack with their code, write past the buffer smashing the stack boundaries, and changing the return address (of the current stack frame) to point back to their code (usually located in the overwritten buffer). Today, this is not possible anymore due to non-executable data protection (DEP) [17], but attackers can still exploit software. Advanced exploitation techniques, based on code reuse, commonly known as Return-Oriented Programming (ROP) [159] and Jump-Oriented Programming (JOP) [34], 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 reuse existing parts of the program and build exploits that can work even when DEP is in place. Thus, memory corruption bugs are still to this day an avenue for attackers to exploit applications, gain control of the system or disclose sensitive information.

In an effort to prevent memory corruption exploitation at its roots, widely used high-level languages, e.g. JavaScript, Rust, etc. rely on memory and type safety. However, modern software development relies heavily on third-party libraries. The heavy use of libraries is particularly common in JavaScript applications [113, 135], and especially in those running on the Node.js platform [207], where developers have easy access to hundreds of

thousands of libraries through the node package manager (npm). The vast majority of the libraries imported in a Node.js application are implemented in JavaScript and thus enjoy the memory and type safety guarantees provided by a high-level programming language and enforced by its runtime environment—at times, augmented with language-based protection techniques [193, 194]. However, not all application operations perform well when written in high-level languages. Thus, it is common to also import a few libraries written in low-level languages or provided only in binary form. These libraries, termed *native add-ons*, implement either functionality not available yet in the pure high-level ecosystem or components that need to be in low-level languages for performance and compatibility reasons. Native add-ons interact with the rest of the program through a thin high-level layers wrapping the library enough to expose runtime-specific naming and calling conventions.

Unfortunately, native add-ons are particularly dangerous to the rest of the application. The complete lack of memory safety means that even a single line of memory-unsafe code may compromise the application's safety and security—that is, *including* those of the (safe and secure) majority of the codebase. Native add-ons can additionally bypass the security guarantees provided by the aforementioned language-based hardening and protection techniques. The exploitation risks of native add-ons compound, as these components are more likely to be targeted by malicious adversaries—exactly because of their vastly higher insecurity and potential impact.

While memory corruption bugs are severe for the security of a system and can be even exploited remotely, there are even techniques that can disclose information to attackers that do not have any authorization to interact with the system. Rather, attackers can disclose sensitive data by launching powerful physical attacks on the system. Besides of the (cold) data stored on secondary storage devices which are usually protected with encryption, sensitive data also reside on main memory (hot data), where they are typically in clear-text. This allows for physical attacks on memory, that can disclose data used during an application's execution. More importantly, it is not only servers or desktops that are under threat. According to [125], more than 40% of business users leave their laptops in sleep or hibernation mode when traveling, leaving their private or corporate data, keys or passwords residing in main memory unprotected. As a consequence, an adversary is able to retrieve data from the main memory, including any stored sensitive data, e.g., session keys, passwords, HTTP cookies, SSL key pairs, gaining this way access to online services, bank accounts, *etc.*. Some of the typical methods adversaries utilize to steal data from main memory are cold boot attacks [87, 89] and DMA attacks [182].

In this dissertation, we focus on techniques which aim to deny the building blocks for successful exploitation of memory vulnerabilities, either physical or remote. In other words, we do not try to eliminate memory vulnerabilities, rather, we aim to prevent their meaningful exploitation i.e., an attacker will not be able to execute arbitrary code or access sensitive data. To accomplish this, we recognise the enhanced security guarantees of

hardware-assisted security mechanisms. Hardware designed for security can be more effective against powerful attacks while minimizing the imposed runtime overhead required by security checks and operations. Contrary to software only design, although a plethora of mechanisms have been proposed and many of them are very effective in defending against exploitation, they usually suffer from two issues. First, the security checks required incur significant runtime overhead. Second, they can be bypassed, since the trusted computing base is broad and attackers can target one of the many *trusted* components, e.g. the Operating System, the firmware, etc.

We divide the work in this dissertation in two categories of techniques aiming to provide solutions for a variety of different security concerns in modern systems. Their common variables are that we utilise hardware to achieve our goals and that we aim to address memory related exploitation techniques. The first, leverages hardware mechanisms already available in commodity off the self processors for security purposes.

Our first work in this category, aims to investigate whether it is possible to achieve memory encryption with common architectural features at a reasonable performance cost. In particular, we proposed the first of its kind software based memory encryption approach based on the `AES-NI` [98] and IOMMU [122] hardware features. Our design ensures that sensitive data will remain encrypted in main memory at all times. Our approach is based on commodity off-the-shelf hardware, and is totally transparent to legacy applications. With this mechanism we aim to protect sensitive information like passwords, secrets, and private data that can be easily exfiltrated from main memory by adversaries with physical access.

Our second work in this category, is the design of a hybrid permission model which aims at protecting managed runtime environments from possibly vulnerable native addons. The permission model is applied all around a native add-on and is enforced through a hybrid language-binary scheme that interposes on any accesses to sensitive resources from any part of the library. A pair of program analysis components infer the add-on's permission set automatically, over both its binary and high-level sides. In order to confine the execution of the third party native module we rely on Memory Protection Keys (MPK), a feature available in the latest Intel CPUs [99].

Our second set of techniques, explores the design and implementation of *in-house* hardware extensions (i.e., extending available open-source processors) and we aim to solve code-injection as well as code-reuse attacks.

Our first approach was the design and implementation of AESASIST: an architecture with both hardware and operating system support for Instruction Set Randomization (ISR). Instruction Set Randomization (ISR) is able to protect against remote code injection attacks, by randomizing the instruction set of each process. Thereby, even if an attacker succeeds to inject code, it will fail to execute on the randomized processor. AESASIST extends a SPARC processor that is mapped onto a FPGA board and runs our modified Linux

kernel to support the new features. In particular, before executing a new user-level process, the operating system loads its randomization key into a newly defined register and the modified processor decodes the process's instructions with this key. Besides that, AESASIST uses a separate randomization key for the operating system, in order to protect the base system against attacks that exploit kernel vulnerabilities to run arbitrary code with elevated privileges. Our design is heavily assisted by hardware and uses AES encryption in order to offer resilience against cryptographic attacks and thus, raise the bar even against Code Reuse Attacks.

Our second hardware extension aims to prevent Control-Flow Hijacking by enforcing Control-Flow Integrity. Control-Flow Integrity (CFI) is a popular technique to defend against State-of-the-Art exploits, by ensuring that every (indirect) control-flow transfer, points to a legitimate address and it is part of the application's Control-flow Graph (CFG). In this work, we explored the implementation of a full-featured CFI-enabled Instruction Set Architecture (ISA) on actual hardware. Our new instructions provide the finest possible granularity for both intra-function and inter-function Control-Flow Integrity. Beyond the academic work, we identified the challenges and lessons learned and we draw the official specification for architecturally supported Control-Flow Integrity for the RISC-V architecture [21]. This was not an easy task, since we had to identify and take into consideration `all` the corner cases in general purpose software and preserve compatibility. In contrast, most work in academia only considers a limited number of corner cases and it is especially hard to be adopted by the industry, without extensive modifications on the original design.

1.1 Thesis Statement

The effectiveness of security mechanisms in thwarting cyber attacks relies on multiple checks as well as the use of many tools (e.g., cryptography). These operations are not cheap and thus impose significant runtime performance overhead. During the design of a mechanism a usual concern is the trade-off between performance and security. Focusing solely on security, often results in substantial slowdown, rendering the rather *secure* application, impractical to use. The end result is that such mechanisms are eventually abandoned. On the other hand, approaches that rely on relaxed policies in order to be practical, often prove ineffective.

One approach that can enhance the security offered as well as minimize the performance impact is the design of hardware that assists the security mechanism. With specialised hardware, many checks can be grouped under a single machine instruction and many primitives (e.g. cryptography) can be accelerated through hardware components. Moreover, hardware is immutable and thus security mechanisms become harder to bypass or disable. However, designing hardware for security is not an easy process and re-

quires careful planning on what parts of the mechanism can be pushed in the hardware domain, considering the circuity area overhead. Beyond hardware additions, the overlaying hardware-software interface must be designed in order for applications to deploy the mechanism. Finally, it is important to preserve functionality and restrict corner cases as less as possible.

This dissertation verifies that *hardware-assisted security mechanisms can successfully prevent a plethora of exploitation techniques while keeping the runtime performance impact within practical limits. We accomplish this, by designing and implementing hardware assisted strategies which prevent memory vulnerabilities from escalating to full-fledged system exploitation.*

1.2 Contributions of this Dissertation

The key contributions of this dissertation are the following:

- We present a memory encryption approach, which ensures that sensitive data will remain encrypted in main memory at all times. Our approach is based on commodity off-the-shelf hardware, and is totally transparent to legacy applications. To accommodate different applications needs, we have built two versions of main memory encryption: Full and Selective Memory Encryption. Additionally, we provide a new memory allocation library that allows programmers to manage granular sensitive memory regions according to the specific requirements of each application. We conduct an extensive quantitative evaluation and characterization of the overheads of memory encryption, using both micro-benchmarks and real-world application workloads. Our results show, that even though our scheme introduces overhead when applied in micro workloads, it imposes low overheads and can be viable in real-world, server applications.

- We design a new hybrid permission model aimed at protecting both a binary add-on core and its language-specific wrapper in the Node.js [142] managed-runtime environment. The permission model is applied all around a native add-on and is enforced through a hybrid language-binary scheme that interposes on any accesses to sensitive resources from any part of the library. A pair of program analysis components infer the add-on's permission set automatically, over both its binary and JavaScript sides. Applied to a wide variety of native add-ons, we show that our framework reduces access to sensitive resources, defends against real-world exploits, and imposes modest overhead.

- We present the design and implementation of AESASIST: an architecture with both hardware and operating system support for Instruction Set Randomization. AESASIST

uses our extended SPARC processor [68] that is mapped onto a FPGA board and runs our modified Linux kernel to support the new features. Our evaluation shows that AESASIST can transparently protect both user-land applications and the operating system kernel from code injection and code reuse attacks. In our design the instructions are encrypted using the AES algorithm. However we keep the overhead within acceptable margins, due to efficiently leveraging the spatial locality of code through modern instruction cache configurations.

- We explore the implementation of a full-featured policy agnostic Control-Flow Integrity(CFI) Instruction Set Architecture (ISA) on actual hardware. Our new instructions provide the finest possible granularity for both intra-function and inter-function Control-Flow Integrity. We implement hardware-based CFI by modifying a SPARC SoC [68] and evaluate the prototype on an FPGA board by running SPECInt benchmarks instrumented with a fine-grained CFI policy. Our design can effectively protect applications from code-reuse attacks, while imposing negligible runtime performance overhead.

1.3 Outline of Dissertation

The report is organized as follows Chapter 2 presents an overview of the attacks we prevent with the strategies presented in this dissertation as well as discuss the efficacy of the counter measures deployed so far. We focus on attacks that target memory related vulnerabilities which can be exploited due to physical or remote access. Chapter 3, contains an overview of various security related architectural extensions that have been included in general purpose CPUs. Chapter 4 presents the design, implementation and evaluation of a main memory encryption scheme that relies on IOMMU and cryptographic instruction set in order to prevent physical attacks. Chapter 5 covers the design and implementation of a Memory Protection Keys (MPK) based framework for isolating the execution of untrusted third-party libraries in managed runtime environments. Chapter 6 describes the design, implementation and evaluation of a cryptographic attack resistant Instruction Set Randomization architectural extension that prevents both Code Injection and Code Reuse attacks. Chapter 7 explores the design, implementation and evaluation of a complete, policy agnostic Control-Flow Integrity Instruction Set that can prevent control-flow hijacking. We also discuss the adaptation of our work in RISC-V architecture. Finally, chapter 8 concludes this dissertation and provides directives for future research.

Chapter 2
Background

2.1 Buffer overflows - Code Injection Attacks

Buffer overflows are one of the most common low-level software bugs exploited by adversaries. During a buffer overflow, a program writes data to a buffer, overruns the allocation boundaries and overwrites adjacent memory location. This behaviour is a well-known security exploit. Code injection attacks enables an attacker to execute malicious code through the exploitation of a software vulnerability. Arbitrary code execution is also possible through the modification of non-control-data [47]. Despite considerable research efforts [57,59,191], buffer overflow vulnerabilities remain a major security threat [48]. Other vulnerabilities that allow the corruption of critical data are format-string errors [55] and integer overflows [197]. These attacks are not limited to only the binary level. Attackers have also exploited vulnerabilities in the input validation of web scripts in order to execute malicious *high-level* code (e.g. SQL scripts, JavaScript, etc.) on the target system.

2.1.1 Defences

Stack canaries [54] is a mechanism, deployed to detect modifications on the stack by placing a unique label before any sensitive data, like the return address, however, an attacker can easily bypass this security feature by overwriting it bit by bit and observing the results. Other attack techniques overwrite the control-flow variables through data pointers that refer to adjacent local variables, effectively avoiding to corrupt the canary value [150]. Code-injection attacks at the binary level have been effectively prevented however through W⊕X policy (i.e. a memory page cannot be writable and executable at the same time). 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 Data Execution Prevention (DEP), which system software leverages in order to mark pages containing the stack and heap as non-executable. In applications written in high-level languages, input sanitization and validation techniques have been proposed in order

to prevent script based code injection attacks [163].

2.2 Return-to-libc

Since Data Execution Prevention prevents the execution of injected malicious code, modern attacks do not depend on injecting malicious code. As a result, attackers sifted the focus to using code already present in executable address space for their nefarious purposes. 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, as well as filling the stack with the appropriate function arguments. 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.

2.2.1 Defences

By deploying Address Space Layout Randomization (ASLR) [187] this type of attack is unlikely to succeed. ASLR is available in most operating systems (Linux, Windows, OSX). This technique randomly shuffles the sections positions 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 over-read. Then, the attacker can calculate the desired function's location using the leaked address.

2.3 Code Reuse Attacks

Code Reuse Attacks (CRAs) rely on executing code already present in the vulnerable application. The most known CRA is Return-Oriented Programming. Return-oriented programming (ROP) is a software exploitation 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 [110] and SPARC [38] It's a more advanced version of stack-smashing, in that it utilizes vulnerabilities that modify the stack in order to overwrite the return address stored within. The new return address is then used to change the control-flow of the executable to the attacker's desired path. Other variances include Jump-Oriented Programming [35] where the attacker overrides function pointers residing in the stack or the heap and Counterfeit

Object Oriented Programming, which is specific to C++ applications [166]. These techniques can be combined in order to amplify the chances of a successful exploitation.

Since the attacker now has control over the stack, and subsequently Control-Flow variables, the execution can be diverted to any address containing code (i.e. is executable). The logical next step is to identify *useful* code and divert the control-flow accordingly. The identified pieces of code are called gadgets.

2.3.1 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 [31]. 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.

2.3.2 Defences

A lot of mechanisms have been proposed in order to counter these types of attacks, however they either impose significant performance overheads or rely on relaxed schemes that only raise the bar for attackers. Safestack [120] separates the data and the return addresses on the original stack, and puts the former in the unsafe stack and the latter in the safe stack. The base address of the safe stack is random, thus the security of the is based on information hiding. However, it has been proven that by forcing an application to spawn many threads and consequently safe stacks an attacker can decrease the entropy of the safe stack addresses [74]. Other than protecting sensitive control flow values from being overwritten other schemes aim to eliminate buffer overflows in binaries. Memory safe languages aim to detect/prevent memory access vulnerabilities. For example, Java runtime dynamically checks array bounds. On the other hand, Rust programming language [93] eliminates possible memory corruptions through static analysis. While memory safe languages are an appealing way to eliminate software bugs, complex low-level software need to be written in assembly or C which are not *memory* safe. Moreover, rewriting every legacy software

and libraries in a new language requires a massive amount of software developing effort.

2.4 Buffer Over-read - Memory Disclosure

Another avenue of exploiting memory corruption vulnerabilities is buffer over-read attacks. Herein, the attacker does not target control-flow variables, rather disclosing memory containing sensitive data. For example, given an application that utilises cryptographic libraries to encrypt network communication, an attacker could exploit a memory corruption vulnerability, read beyond a buffers boundaries and disclose the cryptographic keys of the application [66]. A similar scenario could target session IDs in a web browser scenario.

2.4.1 Defences

A common defence strategy is Intra-process isolation, operating systems focus on process isolation (virtual memory, etc.) to prevent process's from arbitrarily interfering with between them. Intra-process isolation, is required in applications that need to isolate components in the same process. For example, web browsers isolate the execution of different pages in order to prevent malicious pages from accessing sensitive data. A notable family of Intra-process isolation techniques is Software Fault Isolation (SFI), SFI instruments memory operations in order restrict memory access beyond a designated area. Other instrumentation approaches ensure that out-of-bound pointers are transformed into in-bound. Research efforts focus on in-process techniques, that offer isolation guarantees with minimum cost [28, 186].

2.5 Physical Attacks

2.5.1 Cold Boot Attacks

In a cold boot attack [87, 89, 201], the data remanence effect of RAM is exploited by the adversary to extract the data from the memory. There are two ways of achieving this: (i) an attacker can freeze the RAM modules using a refrigerant [85] which then physically removes from the victim's device and inserts them into a device that is capable to read the contents of the RAM; (ii) an attacker can perform a warm boot by running specific attack tools, and retrieve the contents of the residual memory [43]. In this type of side-channel attack, the attacker is able to retrieve encryption keys and sensitive data from a running operating system even when the user session is locked. As has been shown in [172], modern SRAM chips can retain about 80% of their data for up to a minute at temperatures below -20 degrees Celsius.

2.5.2 DMA Attacks

This type of attacks leverage the ability of a DMA interface to allow a peripheral to directly access arbitrary memory regions, and read memory contents without any supervision from the processor or the OS. More specifically, an attacker can program a DMA-capable peripheral to manipulate the DMA controller and read sensitive data stored in memory [58, 181]. This type of attack can be carried out over different IO buses, such as the Firewire, PCI Express or Thunderbolt.

2.5.3 Defences

Markuze et al. [123] leveraged the IOMMU in order to restrict device access to a set of shadow DMA buffers that are never unmapped, and it copies DMAed data to/from these buffers.

Chapter 3
Hardware mechanisms for security

Hardware assisted security mechanisms are not a novel concept, rather, a notion that exists even in very early computing systems. However, in the last decade the number of architectural extensions for security is increasing. This is due to the fact that computing systems are involved in life safety aspects of human lives, as well as the trend to optimise the performance of systems through hardware acceleration. In this section, we present an overview of several hardware-assisted mechanisms relevant to this dissertation.

3.1 Protection Keys

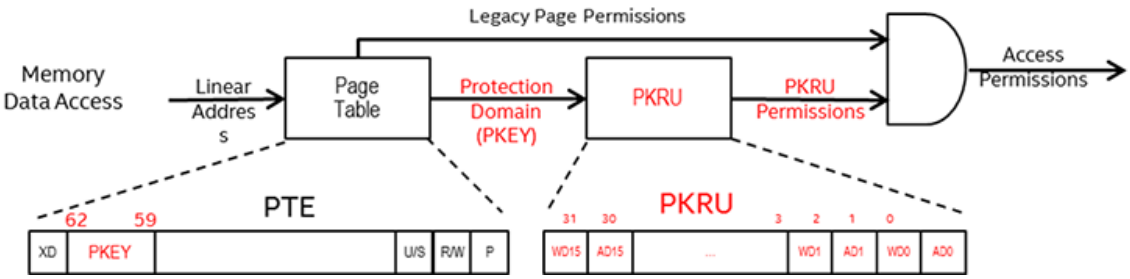


Figure 3.1: Memory access privilege checks when using Memory Protection Keys. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html

Protection Keys is a relatively common architectural feature, first introduced in IBM System/360. Today, IBM storage protection keys [96] are part of Z architecture systems. A protection key is assigned to each virtual page and represents the access authority required for each context. An authority mask register is used for specifying the access rights of each context. IA-64 protection keys [97] are designed to restrict permissions on memory by tagging each virtual page with a unique domain identifier. IBM extended protection keys architecture with 16 Protection Key Registers used as a cache for the access rights

on the protection domains required by a process. During memory accesses, if a key is found during memory translation, it is looked up in the available protection key registers to check the access rights. ARM memory domains [19] offer multiple sand-boxes to a process. There can be 16 memory domains in each process, and a domain access control register (DACR) defines the access rights on each domain. DACR is a privileged register, and thus, domain switches are handled by the supervisor level. In chapter 5 we presented how we leveraged Intel's implementation of Protection Keys [99] in order to isolate the execution of untrusted native library code in Node.js applications.

3.2 Virtual Addressing and memory segmentation

Almost every widely used system relies on virtual addressing. This constrains applications from interfering with each other'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 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 over-read 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.

3.3 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 virtualisation. The operating system always executes at ring 0 (figure 3.2).

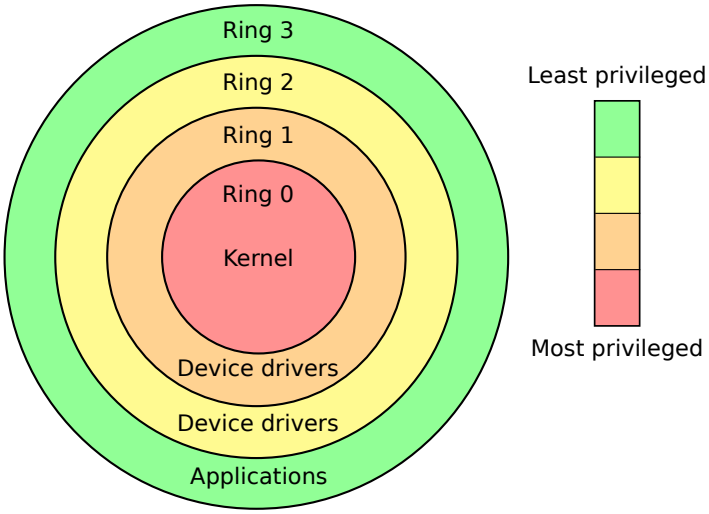


Figure 3.2: Protection rings in x86 processors By Hertzsprung at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=8950144

3.4 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. Code-Reuse Attacks are a sophisticated set of exploitation techniques that can bypass this mechanism.

DEP has different acronyms depending on the processors' brand [2]. 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.

3.4.1 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.

3.4.2 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 [3].

In 2018 GCC and Linux proposed to remove the support for Intel MPX [70, 117]. These decisions were made due to serious concerns regarding the effectiveness of MPX [139]. Even though Intel MPX is a hardware-software co-design the overhead imposed was prohibitive. Temporal memory safety was not provided, thus multithreading was not supported. Compilers had to be designed in order to explicitly synchronise the memory bounds. Several programming idioms cannot be used due to the restrictions on the allowed memory layout. Thus, application developers had to substantially refactor the source code in

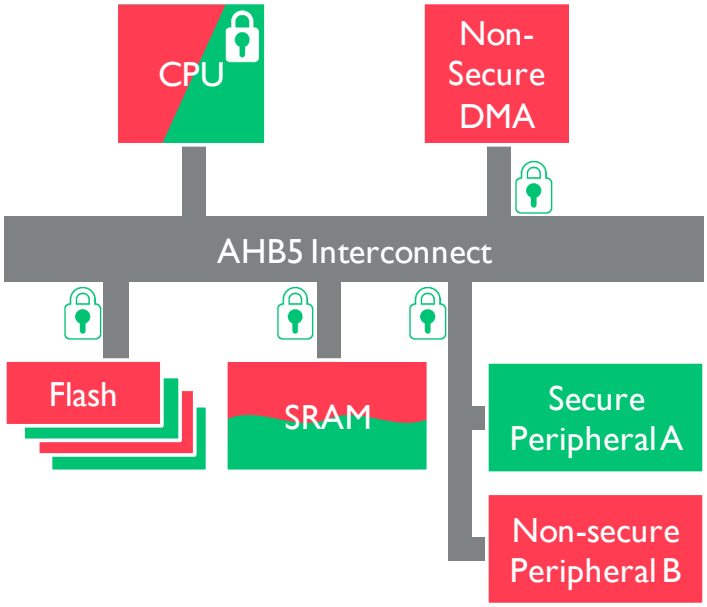


Figure 3.3: Overview of an ARM SoC with trustzone. CPU has both execution domains. Flash and SRAM define trusted and untrusted regions. Peripherals can be defined as trusted or untrusted in case of security concerns.Of course DMA should also be constrained.

order to deploy Intel MPX.

Prior to its introduction in Intel processors, bound checking was a well studied technique in the literature. Software only implementations (e.g. [23], [51]) 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 [64] and CHERI's [203] 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 acceptable margins.

3.5 Trusted Execution Environments

3.5.1 TrustZone

The *TrustZone* feature is available in ARM processors [1]. 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.

3.5.2 Software Guard Extensions

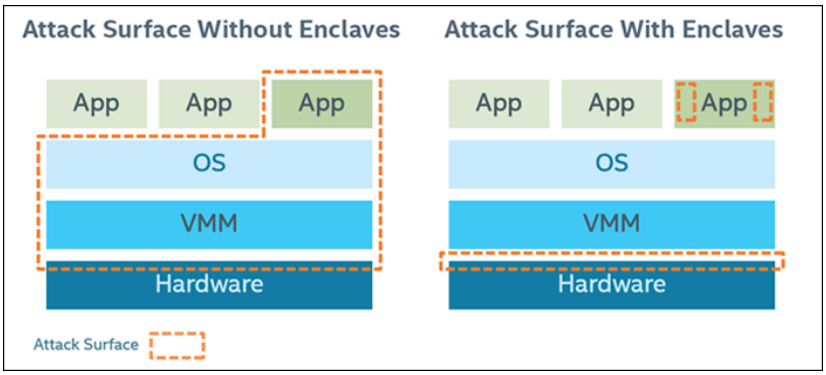


Figure 3.4: Reduced attack surface in Trusted Execution Environments. The protected application only trusts the processor chip. https://software.intel.com/content/www/us/en/develop/articles/intel-software-guard-extensions-tutorial-part-1-foundation.html

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 3.4. 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 to the processor chip, they are decrypted and checked for integrity. The component responsible for those operations is the Memory Encryption Engine [88]. 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. Initially, the size allocated for enclaves was limited to 128MB. The next version of SGX (2.0) supports dynamic memory allocation

and also removes the restriction of 128MB enclave storage size in memory. 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 [16]. 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 ca be untrusted.

3.6 Memory Encryption

In chapter 4 we present how we leverage AES-NI in order to provide full main-memory encryption for application's data. These strategies have been proven effective in thwarting physical attacks and thus, they have been recently introduced with hardware acceleration in commodity off the self processors.

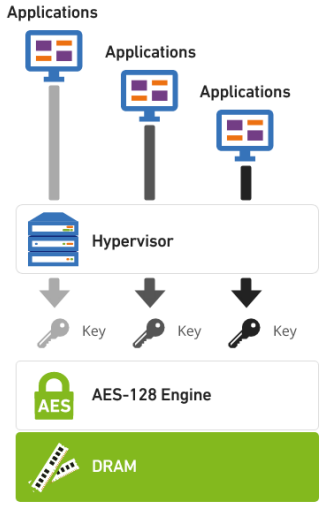


Figure 3.5: Overview of AMDs Secure Encrypted Virtualisation.

3.6.1 AMD EPYC Hardware Memory Encryption

AMD introduced a hardware framework [15] that can provide encryption for in-memory data, at page granularity. The additional hardware includes an AES-128 encryption engine, embedded in the memory controller and a secure processor that is responsible for the key generation and management. Whenever a key is available, main memory data will be encrypted in main memory and decrypted before being loaded in the processor cacheFig. 3.5. These mechanisms can prevent information disclosure from attackers that snoop memory

traffic, or can even launch cold boot attacks in order to retrieve the contents of the main memory.

Another mechanism included in this framework is Secure Encrypted Virtualisation (SEV). Herein, each guest virtual machine is encrypted with different key in order to provide stronger isolation. Each guest OS can indicate which pages should be encrypted 3.5. Finally Encrypted State (ES) mechanism, encrypts the execution state (register contents, etc.) of a VM when it stops from executing. Thus, even a malicious hypervisor, will not be able to retrieve sensitive information. According to AMD this mechanism can even detect malicious modifications of the Encrypted State.

3.6.2 Intel Total Memory Encryption (TME)

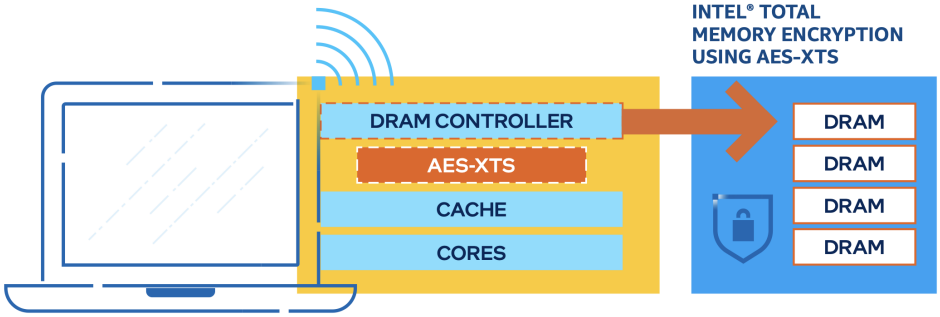


Figure 3.6: Overview of Intel Total Memory Encryption.

In 2019 Intel published a summary of a proposed memory encryption scheme [100]. This architectural extension is very similar to what we present in section (§4) Intel TME is effective against hardware or physical attacks on system memory, for example cold boot attacks. The main memory of the system is encrypted with a single transient key Fig. 3.6. All memory data passing to and from the CPU are encrypted. These include, customer credentials, encryption keys, and other IP or personal information. The encryption key used for memory encryption is generated using a hardened random number generator in the CPU and is never exposed to software. This allows existing software to run unmodified, while better protecting memory. In order to allow benign DMA use scenarios (e.g., PCI devices), Intel TME can be configured through the BIOS to specify a physical address range to remain unencrypted.

,,

,,,,

Chapter 4
Memory Encryption

In this chapter, we present a lightweight, main memory encryption scheme, able to run on off-the-self hardware and support legacy applications. Our implementation is based on Intel's dynamic instrumentation tool called PIN [27], which provides the run-time environment, and supports legacy applications without any code modification. With our approach, application data are always encrypted in main memory, using a 128-bit AES key, which is randomly generated every time the application is launched to make it resistant against key guessing attacks [127]. To cope with the computational overhead of memory encryption, we leverage the Advanced Encryption Standard Instruction Set, which is currently available in the majority of modern microprocessors. Finally, we experimentally quantify the cost of keeping sensitive data secure in practical, real-world scenarios and we respond to the following questions:

- Which are the different possible implementations (dynamic & static instrumentation) and what are the performance implications of each one of them?
- What is the performance cost for the different memory encryption strategies: full & selective memory encryption)?
- What is the power consumption of the different approaches?
- What are the actual overheads of real-world applications, like web or SQL servers, when always keeping their memory contents encrypted?
- How can one reduce the overheads by adopting a more selective/relaxed strategy as opposed to a full memory encryption?

The goal of our approach is to secure *hot data* of running processes by deploying main memory encryption. To achieve this, we use binary instrumentation to ensure that all data will be stored in main memory encrypted.

4.1 Background

4.1.1 AES-NI instructions.

Intel's Advanced Encryption Standard New Instructions (AES-NI) [98], is an extension to the x86 instruction set architecture for microprocessors of Intel and AMD. The main purpose of this instruction set is to improve the speed of applications that perform encryption and decryption using AES. The AES algorithm works by encrypting a fixed block size of 128 bits of plain text in several rounds to produce the final encrypted cipher text. The number of used rounds (i.e. 10, 12, or 14) depends on the key length: 128bit, 192bit, or 256bit. Each round performs a sequence of steps on the input state, which is then passed to the following round. Each round is encrypted using a sub-key called *round key*, which is generated using a *key schedule*. AES-NI can accelerate the performance of an implementation of AES by $3x$ to $10x$ over the traditional software implementation.

The instructions mentioned above work on two operands, the AES state and an AES round key that is used to scramble the state. Each of these instructions performs an entire AES round with a single instruction, exclusively on the processor without involving RAM. Instead of using memory locations for these operands, the AES instructions work on XMM registers. On 64-bit systems there are sixteen 128-bit XMM registers `xmm0` to `xmm15`. AES states and round keys fit exactly into one XMM register as they are 128 bits long. This way, AES-NI eliminates the need for AES lookup tables, which have been a source of cache-related timing side channel vulnerabilities [143, 190].

4.1.2 Key schedule.

To defend against cryptanalysis, algorithms like AES and other block ciphers employ a concept called *key schedule* in which a different key is used for each round of encryption. Particularly in AES, only the first round uses the original secret key; each of the subsequent rounds uses a different key that is generated every time by permuting the previous rounds' key. Traditionally in software AES implementations, all 20 keys of the key schedules (encryption and decryption) are precomputed and stored in RAM for performance purposes. However, in our case such approach would allow local attackers to easily exfiltrate the keys from the memory by employing physical attacks (e.g. a cold boot attack).

4.1.3 Intel PIN

In this work, in order to keep the data residing in main memory always encrypted we instrument the memory access- ing operations and we enhance them with the appropriate AES-NI instructions. To implement this instrumentation, we use the execution environment of the Intel's dynamic binary instrumentation tool PIN [27]. We chose to use this tool due to its high-versatility and support to multiple architectures (x86, x64, ARM, and more).

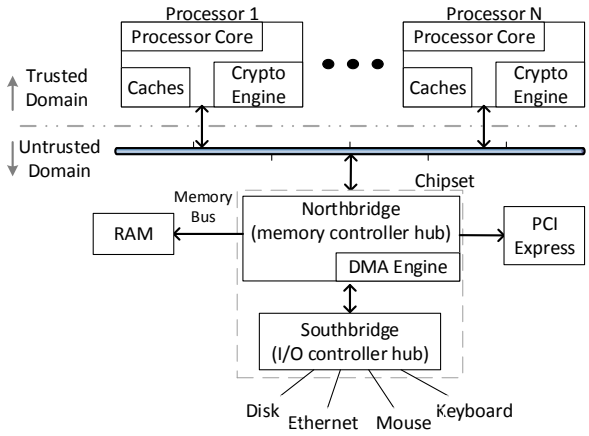


Figure 4.1: Data are always encrypted when residing in main memory or moving between the different components of the untrusted domain.

In addition, PIN enables the developer to inspect and tamper an application's original instructions and operates entirely in user space. It disassembles the original instructions of the application and compiles them along with the instrumentation in a just-in-time manner. The instrumented code is then stored and executed in a code cache structure.

4.2 Threat model

In the literature the typical threat model for data memory encryption involves hardware and/or software attacks. In this work we assume attackers that aim to steal secret information. These hardware attacks may be motivated by financial gain, e.g. capturing credit card PIN numbers, corporate espionage, intelligence stealing from government or military infrastructures, etc.

Our main assumptions, which are in line with the related literature [92], are that the processor provides a secure region, within which sensitive information can reside. As we see in Figure 4.1, all components outside of the processor are assumed to be vulnerable, including RAM and its interconnections, like the data and memory bus, I/O devices, etc. We assume however that the operating system kernel is trusted. This is a reasonable assumption, since an adversary capable of controlling the operating system can cause more significant damage than just eavesdropping sensitive data. The core idea of main memory encryption related techniques is to avoid potential data breaches, and make any adversary with the above properties unable of observing, deleting, replacing or modifying any piece of data existing in a victim system.

4.2.1 In-Scope Threats.

We assume that the adversary has physical access to the victim's system where sensitive information is stored, and that the machine can be exposed to physical hardware attacks, like cold boot attacks ,bus monitoring attacks, and DMA attacks.

4.2.2 Out-of-Scope Threats.

Apart from the above attacks, obviously there are many more threats for the data residing in memory, that fall outside the scope of this paper.

Memory disclosure attacks.

This type of attacks aim to compromise the software, accessing this way possible secrets and passwords. Such attacks exploit a software vulnerability to install malicious code. Although this type of attacks are quite common and important to consider, this paper focuses on attacks that do not rely on running compromised software.

Side-channel Attacks.

Such type of attacks aim to extract sensitive information by exploiting physical properties (like timing information or power consumption) of the cryptographic implementation. These attacks usually have a limited accuracy and require a relatively high level of sophistication, especially when the attacker cannot run arbitrary code on the device, therefore they fall beyond the threat model of this work.

Sophisticated Physical Attacks.

It is hard to defend against every type of physical attacks. Indeed, there are several Advanced Persistent Threats (APTs), usually deployed for corporate espionage, intelligence stealing from governmental or military infrastructures etc., which under specific circumstances, can achieve severe data breaches. However, such attacks require specialized equipment and can often take several months even when carried out by a skilled attacker.

4.3 Main Memory Encryption

In this section, we describe our main memory encryption strategy, for securing the data of running applications. We instrument the load and store instructions of several benchmarks and real world applications. By using the provided rich API of PIN, we instrument the original load and store instructions of an application's binary. Specifically, we insert a callback to intercept each of the original instructions and we instrument the ones that

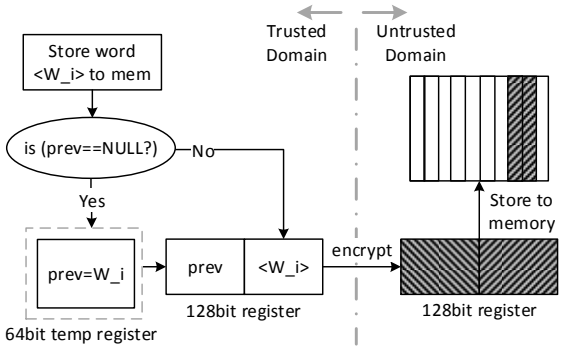


Figure 4.2:
Subsequent store instructions have words encrypted as a bundle in the same block and are then stored on main memory.

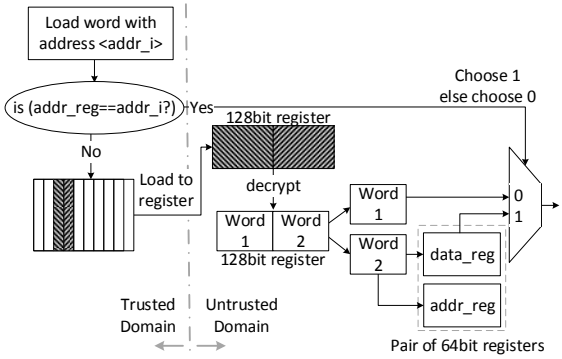


Figure 4.3:
For sequential memory accesses, the block is decrypted once and the 2nd word is retrieved from the same register instead of re-decrypting the same block.

load or store data from or to main memory respectively. After intercepting these instructions we extract the data from the utilized register and, by leveraging the described above AES-NI instructions, we apply encryption or decryption depending on the instrumented instruction. Finally, the outcome of these cryptographic operations replaces the original register's value and the instruction is ready to continue its operation.

The instrumentation of load and store instructions can be implemented in two ways, either (i) *statically*: by instrumenting the memory operations on the binary level, or (ii) *dynamically*: by running the corresponding binary executable on top of a dynamic instrumentation tool.

The static instrumentation of the binary executable offers better performance, however requires the static instrumentation of all linked shared libraries as well. On the other hand, dynamic instrumentation is able to handle complex run-time code manipulation cases, such as dynamically generated (JIT), obfuscated or even self-modifying code. Thus, even though dynamic instrumentation has an extra performance overhead (as we will see in Section 4.4), it is considered more flexible and supports both shared libraries and run-time generated code.

4.3.1 Full memory encryption

An important design decision, when applying memory encryption, is how to encrypt the data. In 64-bit architectures memory operations operate up to 64-bit words. However, the AES algorithm operates in block units, where each block is 128 bits. Hence, during each memory operation we need to collect 128-bit aligned data. This is accomplished by mak-

ing use of two `xmm` registers, one as a load buffer and another as a store buffer. In case of multiple encryptions, this register helps us temporarily keep data until the next store instruction targeting adjacent data is issued. The sequential store instructions, as can be seen in Figure 4.2, will get a couple of words encrypted in the same block and then the block will proceed to be stored in the main memory. In case of decryption, this register allows us to *pre-fetch* data during sequential memory accesses. This way, as seen in Figure 4.3, when a process loads a word and then loads its very next one, it will retrieve it directly from the register instead of decrypting again the same block. This solution has the additional benefit of hiding some decryption latency when consecutive words are accessed.

We note that it could be possible to operate on multiple blocks (> 128 bits). This approach may benefit from less encryptions and decryptions in sequential memory accesses, which would improve the performance of programs that exhibit cache locality and reduce their overall execution time. However, it would also require quite extensive buffering, which would result to massive utilization of registers. The data will need to be in the registers for an unknown period of time, until they reach the proper size of the block. More importantly, applications will have performance gains solely in the case of sequential data accesses, while in the case of random memory accesses, a large part of the decrypted data will remain underused and will be quickly evicted from the registers. Using the above encryption scheme, all data placed in the memory are encrypted, however they are still not well-protected. Given that each block is encrypted separately, an attacker is able to identify identical cipher-text blocks that yield identical plain-text blocks, after scanning the entire memory. These unprotected data patterns allow trivial attacks available in the adversary's tool-chest even in the single-snapshot scenario of the cold boot attack. To remedy this issue, we use a stream cipher encryption mode of operation instead of block cipher. In our case, since applications may need to randomly access non-sequential single blocks need to be decrypted separately. To obtain this random access property during decryption, we employ the CTR mode of operation by using a per-session random nonce and a per-block counter. This way, we turn the block ciphers into a stream cipher, eliminating the potential appearance of patterns.

Handling system calls.

Applications often perform specific operations that only the operating system's kernel has the privilege to execute. For instance, hardware-related operations (e.g. accessing a hard disk drive), or communication with integral kernel services, such as process scheduling. The request of such privileged applications (i.e. system call) usually is followed by application user data and parameters that need to be passed to the kernel. In our case, all of the data passed from user to kernel space are encrypted. In order to access the data, the

kernel obtains the proper process key (see Section 4.3.4 below), and decrypts the parameters before and respectively, encrypts any results after executing the system call. There are system calls that are so frequently used from user-space applications, that can dominate the overall performance. To avoid the expensive performance penalty of system calls and context-switches, the kernel uses a virtual dynamic shared object (`vDSO`) mechanism. In particular, selected kernel space routines (e.g. `gettimeofday(2)`) are mapped into the address space of user-space applications by the kernel, enhancing thus the performance of these applications. Given that there is no switch to the kernel space, in our case, `vDSO` is treated like any other shared library object i.e. having its store and load instructions cryptographically instrumented.

Signals and non-local jumps.

Another case we need to take into account is signals. When a signal arrives at an application, the used registers and the processor's state must be stored for the execution to smoothly continue afterwards from the current state. In our case, the specific registers may contain sensitive data that we cannot risk to be spilled in memory in plain-text. To overcome this, we modified the kernel by using the proper process key, to encrypt their contents before saving them to `sigcontext` structure. When the specific execution continues, the loaded values are decrypted before restored back to the registers. In a similar manner, we deal with the case of non-local jumps (i.e. `setjmp/longjmp`). Specifically, in case of `setjmp`, the data from the utilized general purpose registers are being encrypted before stored in a jump buffer in memory. On the other hand, in case of `longjmp` the data are decrypted after being restored from the jump buffer to the registers right before the application jumps to the return address set by the `setjmp`.

Handling context switches.

Typically the CPU loads data at run-time in its registers in order to perform its computations. When context switch take place, all the previously used data from the registers are moved onto the stack, which resides in main memory. Considering that there are cases where this data may be sensitive, sensitive information may be present unencrypted in main memory, if these evictions are left unhandled. In our case, these evictions may swap out to sensitive states of AES stored in `XMM` registers, even though they were implemented to run solely on CPU. We modified the kernel's typical context switch procedure to encrypt the content of `XMM` registers before they get evicted and decrypt them after the process is switched back. We achieve this by encrypting and decrypting the contents, right before and right after `FXSAVE` (i.e. store to register) and `FXRSTOR`(i.e. restore from register) instructions respectively.

4.3.2 Selective memory encryption.

Having all memory encrypted provides the best protection for all applications. However, our experiments in Section 4.4 show that the overhead, in terms of performance, can increase significantly. To lower the performance overhead of FME, we propose the encryption only for memory regions that contain secret or sensitive data. This approach could result in much lower overheads during execution, proportional to the size of the data that need to be protected from memory attacks. Unfortunately, though, the exact location of sensitive data in memory is very difficult to be known in advance. Instead, it will require the developer to define the exact memory regions, the sensitive data will later reside in. One solution would be to use `#pragma` directives to provide additional information about which variables will be encrypted. However, this would restrict memory encryption to static variables only and does not offer much flexibility to the developer. To address this issue, we implement a secure memory allocator, namely `smalloc`, to dynamically allocate arbitrary size of memory from the heap. In order to have an integral number of blocks, the memory is allocated in multiples of 128 bits. Any data written in this portion of memory allocated with this allocator will always be encrypted. To achieve this, `smalloc` taints the memory regions it allocates to ensure that the corresponding memory addresses have to be encrypted or decrypted when accessed. For instance, during load operations we can determine if the loaded data originate from `smalloc` and thus must be decrypted before being read. The metadata of `smalloc` are a structure for each allocation to note the starting memory address that the segment begins along with its allocated size to denote the total length of the tainted area.

De-allocated memory pages

Memory pages that have been de-allocated after being allocated by an application handling sensitive data may contain left-overs. These may be readable by attackers, enabling them to retrieve parts or even the entirety of the sensitive information. Even though Linux has a kernel thread responsible for zeroing-out the freed pages, due to internal performance optimizations, there is no guarantee when this will occur. In traditional systems this could pose privacy risks as one may get to read a region of memory that contains sensitive data. In our case, after memory allocation, all sensitive data are encrypted before being placed in the heap. Therefore, sensitive memory disclosure is not possible, as an adversary will read cipher-text bytes.

4.3.3 Protecting memory from illegal access

After ensuring the confidentiality of written-through data, a problem that may arise is the case of DMA attacks, where it is possible not only to read data from memory, but also to

write. As a consequence, an attacker could inject the OS with malicious code. To mitigate this issue, leverage the Input-Output Memory Management Unit (IOMMU) [122]. The major x86 ISA manufacturers ship their CPUs with this feature supported (see Intel VT-d [11] and AMD-V [12]) The IOMMU is an IO mapping mechanism, which translates device-visible virtual addresses to physical addresses using, OS-provided mappings. Besides, it also provides memory protection, where memory is protected either from malicious devices that attempt to perform DMA attacks or from faulty devices that initiate errant memory transfers. This protection is achieved by enabling the OS to restrict *who can access what memory region*. As a result, a device cannot read or write to memory that has not been explicitly allocated or mapped to it. In our case, IOMMU is properly configured in order to forbid any access to in-memory kernel or application data. After the DMA transaction is complete, the OS destroys the mapping, restricting further access to the device. Since the OS controls the DMA mapping, sensitive physical memory addresses will not get mapped for the DMA, preventing an attacker from reading or writing.

4.3.4 Key Management

In this section we describe how we protect the AES secret keys that are used for encryption and decryption, against all attacks within our threat-model (described in Section 4.2). Previous works have shown that it is sufficient to prevent sensitive data and algorithmic state from leaking to RAM by implementing the cryptographic operations using on-chip memory [50, 129].

In our approach, each process is assigned with a different key, that is stored in the Process Control Block (PCB) structure. The PCB contains all the information needed to manage a particular process, and is placed at the beginning of the kernel stack of the process. Still, since the kernel memory is vulnerable to cold boot attacks, each *process key* is encrypted before it is stored in the PCB. The process keys are encrypted using a master key which is stored, similar to Tresor [129], inside a pair of debug registers. We modify the kernel to utilize these debug registers and, by using the master key, encrypt/decrypt the appropriate process key, avoiding to store any key in main memory and possible side channel attacks. The reason we utilize debug registers is that, by default, they can be accessed only from ring 0 privileged level. Thus, they cannot be reached by malicious user-level applications and more importantly, they are not used in procedures like context switch, setjmp/longjmp or signal handling. Additionally, we have modified the `ptrace` system call to respond with `EBUSY` error to any application that may request the particular registers, preventing them from being accessed from user level.

We need to note at this point, that there are studies questioning such use of debug registers to store secrets [32]. An attacker is able to inject and execute code in ring 0 privilege level by deploying a DMA attack, and consequently, disclose the secrets stored in the de-

bug registers. In our case, by leveraging IOMMU, we prevent illegal memory injections and as a result, we eliminate the possibility of such attacks.

Bootstrapping

The master key that encrypts the process keys inside PCB is randomly generated at boot time. To achieve that, we modified the kernel to use `RDRAND` instruction to perform on-chip hardware random number generation and create the next master key. The new master key must be present to every core of the processor. The core responsible for the master key generation (core with ID `0`), also distributes it across the rest of the cores through the Memory Type Range Registers (`MTRR`), which are visible to all cores. The rest of the cores will spin on a shared variable till core `0` sets the value to true denoting that the new key has been generated and placed in the `MTRR`. After that, each core can obtain the key, store it in its local debug registers, and finally increment atomically a shared counter. By monitoring this shared counter, core `0` knows how many of the cores have acquired the new key. When all cores have the master key, it immediately cleans the key from the `MTRR` registers and the boot process continues normally.

There are cases where data from the memory need to be swapped-out from memory and stored in the disk. Such data, would not be able to get decrypted after boot if it gets stored encrypted with the current master key. In these cases however, we assume that the users have deployed not only memory encryption but also full disk encryption (FDE). This means that the data will get encrypted with the FDE's key, before being swapped to disk.

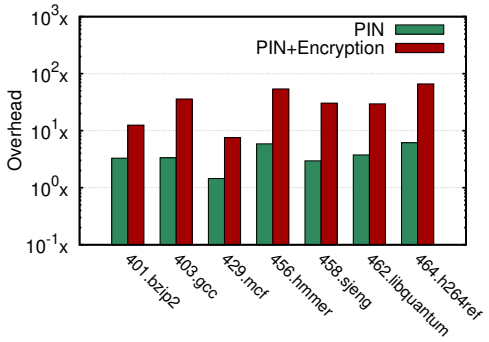
4.4 Performance Evaluation

In this section, we evaluate the performance of both our proposed main memory encryption strategies: Full Memory Encryption (FME) and Selective Memory Encryption (SME). Below we present our methodology and our setup, as well as our findings. To perform the experimental evaluation we use two different hardware platforms: For the performance measurements we used a server that is equipped with two six-core Intel Xeon E5-2620 operating at 2.00GHz, with 15MB L2 cache each. The server contains 8GB RAM and an Intel 82567 1GbE network interface.

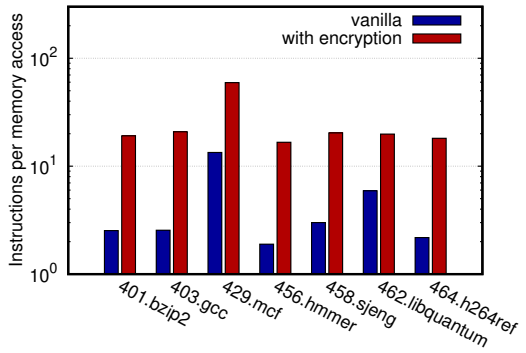
4.4.1 Full Memory Encryption evaluation

At first, we measure the overhead imposed for encrypting all data stored in main memory. This evaluation, determines the cost of the most intensive but secure strategy, where every single byte written in memory is encrypted and respectively decrypted before it is read.

Regarding our memory encryption approach, we insert callback functions to PIN, thus intercepting each of the original instructions and we instrument the ones that either load



(a) Runtime overhead of dynamic instrumentation using the SPEC suite.



(b) Number of instructions per memory access with and without memory encryption (vanilla).

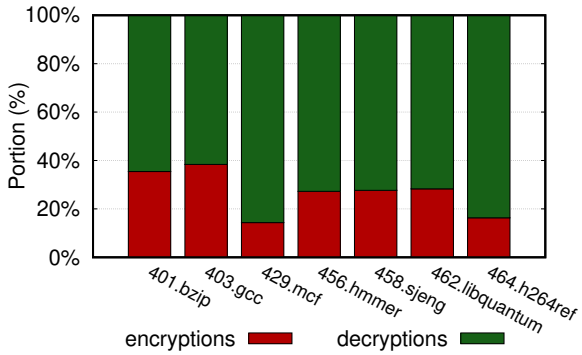


Figure 4.5: Portion of cryptographic operations in each SPEC benchmark.

or store data from or to the main memory. We then extract the data from the utilized register and we apply encryption or decryption depending on the instrumented instruction. Both encryption and decryption are performed using the AES-NI instructions. The output of these cryptographic operations replaces the original register's value and the program continues its execution to the next instruction.

To evaluate the performance of our approach along with the overhead imposed by the binary instrumentation, we measure the performance of (*i*) a vanilla application (listed as Native), (*ii*) a dynamic instrumentation of the application's store and load instructions using PIN (listed as PIN), and finally (*iii*) our approach: encryption with AES using dynamic instrumentation (listed as PIN+Encryption). To measure the plain instrumentation overhead produced by PIN (case (*ii*)), we perform memory instruction instrumentation with empty function calls, instead of any cryptographic operation.

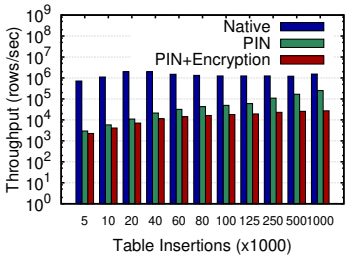
4.4.2 Benchmarks

In the first experiment, we measure the performance of FME using several representative benchmarks, extracted from the SPEC CPU2006 suite (CINT2006). These benchmarks are comprised of several computational and memory intensive applications aiming to stress both CPU and main memory usage. In Figure 4.5, we see the portion of cryptographic operations in each benchmark and in Figure 6.6, the slowdown of a simple dynamic instrumentation of the application's load and store instructions (PIN). This number provides us with a baseline for the overhead introduced by the PIN tool. In the same figure, we present the results of the instrumentation with the appropriate AES-NI instructions to encrypt or decrypt every chunk of memory that is stored in or loaded from the main memory (PIN+Encryption).

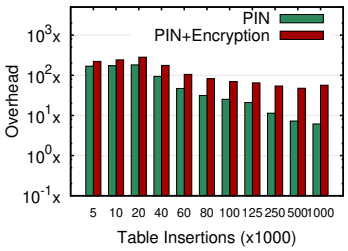
As we observe, the run-time overhead of simply instrumenting the application's load and store instructions reaches up to 6 times slowdown for `h264ref` benchmark, while the additional overhead when adding encryption reaches up to 10 times slowdown. The major slowdown in performance arises from the fact that the data are encrypted and decrypted even when residing in the cache. As the caches in `x86` architecture are not addressable, data can reside there in clear-text, without the concern of being leaked. Unfortunately, as it is not possible to check if specific data are cached or not, we cannot benefit from memory locality. In Figure 4.4b, we measure the instructions per memory access with and without our memory encryption approach. The average encryption cost is an additional 14-18 instructions. This number is not constant; it depends on the benchmark's synthesis of memory accesses and how sequentially the data are being accessed. Due to our prefetching mechanism (described in Figures 4.2 and 4.3): (i) in case of store, encryption takes place only every 2 words (load from register previous word and encrypt the pair - 28 instr.), when (ii) in case of load, the word can be fetched directly from register (8 instr.) or retrieved after decrypting a block (then the unneeded second word has to be stored in the register - 26 instr.).

4.4.3 Real-world applications.

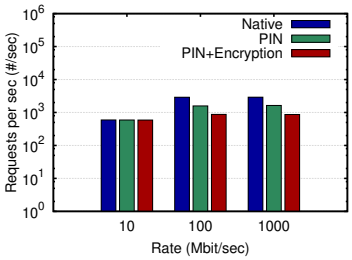
Additionally, we evaluate our approach in a real scenario using two real-world applications. The first, is the SQLite3 relational database management system. We used the C/C++ SQLite interface to implement a simple benchmark that reads a large, 60 MBytes, tab-separated file including 1,000,000 rows of data and updates a table's entries with the respective values. Figure 4.6a shows the achieved throughput, while Figure 4.6b shows the slowdown when inserting data into the database as a function of the number of insertions. As expected, the more rows the benchmark updates, the higher the imposed overhead becomes, since the number of memory encryptions increases. In contrast to that, the cost of the instruction instrumentation (PIN) is always proportional to the number of the table



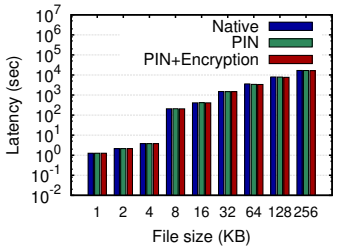
(a) Achieved throughput when inserting 1M rows into the database.



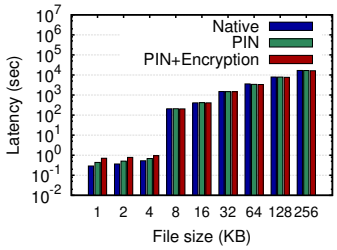
(b) Slowdown when inserting 1M rows into the database.



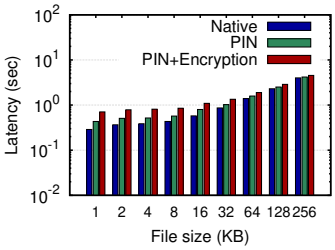
(c) Req/sec when downloading a file of 1522 bytes using different transfer rates.



(a) Client is over a 10 Mbps network.



(b) Client is over a 100 Mbps network.



(c) Client is over a 1000 Mbps network.

Figure 4.7: Average latency per request when downloading different files from a Lighttpd web server as a function of the requested file's size.

insert instructions, resulting to almost linear overhead to the application.

As a second real-world application, we ran the Lighttpd web server both as a vanilla system and with the two versions of dynamic instrumentation In the first experiment, we used a separate machine located on the same local network to repeatedly download a file of 1522 bytes. We synthetically limit the rate of the client's network line to three different network transfer rates: 10, 100 and 1000 Mbit/sec. As can be seen in Figure 6.8, when the bandwidth for the client is 10 Mbit/sec, the memory encryption overhead is almost hidden by the network latency. As a result, the user faces a negligible slowdown of 0.17% for having FME enabled when the cost for the instrumentation is an additional 0.4%. On the other hand, the corresponding overhead for encryption at the higher rate of 1000 Mbit/sec reaches up to 43.7%. Our results indicate that in real-world applications over the Internet the cost for keeping a web server's memory fully encrypted is practically tolerated.

We conduct follow up experiments modifying the usage scenario in the following way. We use the same machine and the same three different network transfer rates to repeatedly download 9 files of different sizes, ranging from 1 KB to 256 MB. We then measure the average requests per second performed for each file. To make this experiment as realistic

as possible, we use the most representative workloads found in production web servers. Such workloads include queries for short snippets of HTML (about 1 KB), e.g. user updates in micro-blogging services like Twitter or Tumblr, or portions of articles found in wikis (2.8 KB on average). Other workloads include photo objects of 25 KB size on average, used in photo-sharing sites that serve thumbnails. In general, as reported in [65], the most common file size is between 2-4 KB and regard HTML files, while 95% of all files are less than 64 KB in size. In Figures 4.7a, 4.7b and 4.7c we present our results for the same experiment in the network transfer rates used above: 10 Mbps, 100 Mbps and 1000 Mbps. We immediately notice that in the case of 10 Mbps, the slowdown introduced from the memory encryption is close to zero, regardless the size of the downloaded file. In case of higher rates (i.e. 100 and 1000 Mbps) we observe that bigger files produce higher latency and as a consequence, hide the memory encryption cost. The average performance overhead imposed by encryption as calculated from the results in Figure 4.7 is 17%.

4.4.4 Static Instrumentation

The alternative of dynamic instrumentation is to statically parse the executable and instrument the load and store instructions. Although this approach requires the instrumentation of all linked shared libraries as well, it is able to provide significantly better performance. In the following experiment, we measure the performance, and more specifically, compare the execution time of the two different approaches. We use a very simple application which copies an array of 512 MB size, along with two secure versions of it: The first version, encrypts the array contents before storing them on main memory by dynamically instrumenting the store and load instructions using PIN. The second one statically encrypts the array's cells by utilizing in-line AES-NI assembly instructions. In the first two columns of Table 4.1 we can see the execution time of each approach as well as the imposed latency overhead compared to the unsecured native application and its binary instrumented version respectively. We observe, that the application with the dynamically instrumented encrypt/decrypt operations on the load and store instructions is 9.56 times slower than the plain instrumentation. Additionally, the static memory encryption makes the application 4.29 times slower compared to the insecure version.

Next, we statically instrument the same benchmarks of SPEC suite as previously and we perform main memory encryption measuring again the run-time overhead. In figure 4.8, we compare the overhead imposed by static and dynamic instrumentation and also the performance improvement of the use of pre-fetching in both cases. As expected static instrumentation performs better (almost 1.7x) than dynamic. In addition, we see that our pre-fetching mechanism, by reducing the number of cryptographic operations in sequential memory accesses, significantly reduces also the performance of our approach

Table 4.1: Encryption cost in the two implementations, in terms of execution time and power consumption.

	Type	Execution Time (sec)	Overhead	Energy Efficiency (Joules/mbit)	Overhead
Dynamic	PIN	2.064999	-	0.03983	-
	PIN+Encryption	19.73596	9.56x	0.52276	13.12339x
Static	Native	0.406917	-	0.00849	-
	Native+Encryption	1.745001	4.29x	0.03072	3.61776x

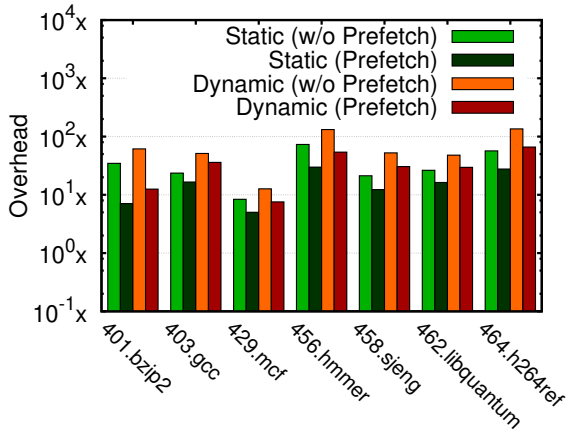
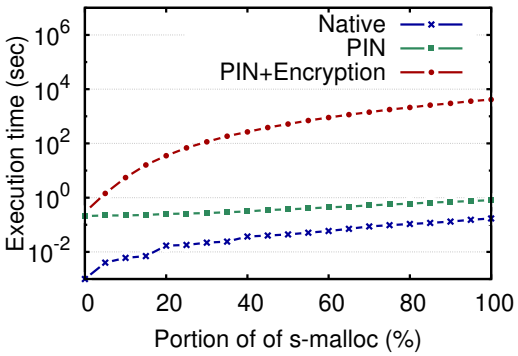


Figure 4.8: Overheads of static and dynamic instrumentation with and without prefetching for the different benchmarks.

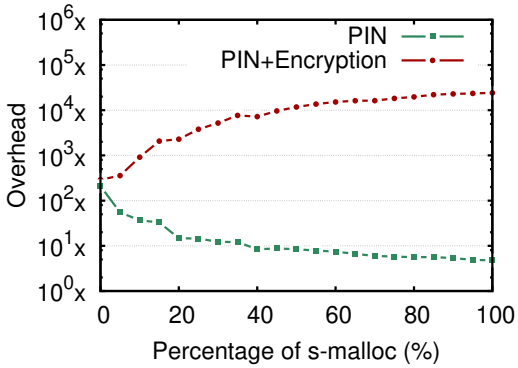
(4.9x on average).

4.4.5 Selective Memory Encryption

Contrary to Full Memory Encryption one may prefer to follow a more Selective Memory Encryption (SME) strategy to reduce the imposed overhead. To evaluate this strategy, we implemented a `smalloc` prototype, to explicitly mark some data as sensitive and only encrypt this data before storing them to memory. Additionally we created a custom benchmark which copies different sized chunks of data from a large array to the heap. Figure 4.9a shows the results for execution time as a function of the portion of data considered as sensitive. The execution time when native applications use `smalloc` without instrumentation increases with the percentage of sensitive data. On the other hand, the cost of instrumentation is also increasing but not as rapidly since it does not depend on the data being stored in memory. Hence, as can be seen in Figure 4.9b the instrumentation overhead

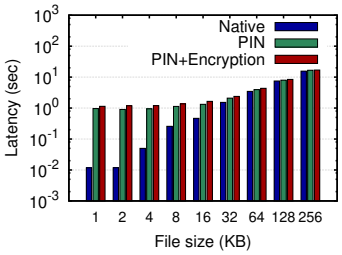


(a) Execution time as a function of the portion of sensitive data.

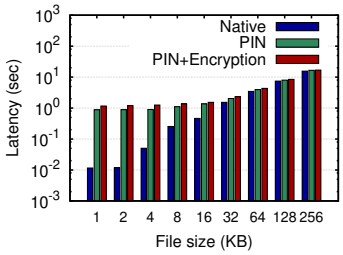


(b) Overhead over native as a function of the portion of sensitive data.

Figure 4.9: Storing different portion of an array's data to the heap. Data considered as sensitive gets encrypted before sent to main memory.



(a) Client is over a 10 Mbps network.



(b) Client is over a 100 Mbps network.

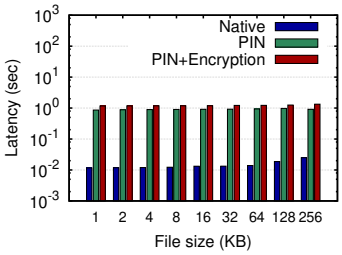


Figure 4.10: Client is over a 1000 Mbps network.

Figure 4.11: Average latency for performing an SSL handshake during a client's connection to a web server where the latter's private key is considered as sensitive.

over native is actually decreasing as the percentage of data increases. Furthermore, the overhead caused by the memory encryption follows a logarithmic growth with the increasing percentage of data being encrypted. Thus, in case of a chunk of data including 10% of sensitive information, the cost to guarantee its confidentiality is latency 24.90 times higher than the unencrypted case.

In our macro-benchmarks, we used the Lighttpd web server as a real world application and the popular Apache HTTP server benchmarking tool of ApacheBench (ab). Web services are a good case of a single physical machine serving multiple users who need to be assured that sessions will be secure during their online transactions. Consequently, we can state that the keys used from the web service during the HTTPS protocol are highly

sensitive, and in need of protection against unauthorized access.

4.5 Related Work

There are various approaches proposed, implemented either in software or hardware, aiming to defend against cold-boot attacks in both academia and industry.

Halderman et al. described cold boot attacks [90], and also discussed some forms of mitigation. Mitigations included deleting sensitive data and keys from memory when an encrypted drive is unmounted, obfuscation techniques, and hardware modifications such as intrusion-detection sensors or encased RAM. However, the authors, eventually admit that these solutions do not constitute complete countermeasures, applicable to general-purpose hardware.

Nagarajan et al. [130] propose compiler-assisted memory encryption for embedded processors, based on a number of assumptions and some limited hardware support. The proposed compiler, supports memory encryption by introducing special instructions to calculate One Time Pads prior to loads and stores. In addition, it assumes the existence of additional process-unique registers used to store the counters. Memory areas for the unique key and global counter is also provided inside the CPU as well as the existence of a cryptographic unit. In [206], the authors assume a powerful attacker with physical access to the machine and able to launch DMA, bus snooping and cold boot attacks in order to disclosure sensitive data residing in the main memory. Their approach focuses on encrypting sensitive data and code residing in the main memory and decrypting and locking them when moved in the cache. Contrary to our approach, their work is tightly woven with ARM specific features, e.g., cache locking and TrustZone.

Towards the same direction, Sentry [50], uses ARM-specific mechanisms in smartphones and tablets to keep sensitive application code and data on the SoC rather than on DRAM. They observe that sensitive state data only need to be encrypted when the device is screen-locked. Consequently, Sentry decrypts and encrypts the memory pages of sensitive applications as they are paged in and out, thus avoiding leakage of sensitive information to DRAM when the device is screen-locked. AESSE [128] was designed to provide Full Disk Encryption (FDE) and protect the required keys by storing the encryption key in the Streaming SIMD Extension (SSE) registers of the CPU, while access to these registers is disabled for user-level code. The authors however, admit that many common applications (like multimedia applications e.g. OpenGL) heavily utilise SSE registers and therefore there is a significant collision with AESSE. TRESOR [129] is a kernel module and successor of AESSE. Instead of the SSE registers it utilizes the debug registers to store the encryption key. In addition, similar to our approach, it leverages AES-NI instruction set to eliminate cold boot attacks achieving this way far better performance than the AESSE.

Loop-Amnesia [170], similar to TRESOR, is a kernel-based disk encryption mechanism

which aims to eliminate cold boot attacks. To achieve this goal, it leverages on the ubiquity of model-specific registers (or MSRs) of modern CPU architectures, to store the encryption keys there rather than in RAM. Thus, data are unreadable to a perpetrator of a cold-boot attack. Currently however, it does not support the AES-NI instruction set.

PrivateCore's commercial product, namely vCage [153], relies on a trusted hypervisor to implement FME for commodity hardware by executing guest VMs entirely in-cache and encrypting their data before they get evicted to main memory. Although it is more cloud-oriented, vCage shares with our approach similar resistance to the same type of physical attacks.

Trustwave's BitArmor [124], is a commercial solution that claims to be resistant against cold boot attacks. BitArmor aims to shield the system as soon as abnormal environment conditions are detected. More specifically, it uses temperature sensors and in case a sudden temperature drop is detected it initializes a memory wiping process. As demonstrated in a recent study [86], this approach raises the bar, but it cannot prevent the attack.

In [183], the authors propose an architecture for a Trusted Processor Model (TPM). This architecture includes a monolithic processor, which integrates several architectural mechanisms into a conventional architecture. Their approach includes memory integrity verification on off-chip data, memory encryption, and secure context management. Regarding memory encryption, it uses AES to encrypt and decrypt off-chip memory on a L2 cache block granularity. Finally, Intel provides processors with Software Guard Extensions (SGX) [101]. These extensions enable applications to encrypt specific data by placing them inside secure memory regions, called enclaves. The data that reside in enclaves are protected even in the presence of highly privileged malware. SGX however, requires complete redesign of an application in order to be leveraged (e.g., separate the safe and unsafe parts). However, our approach can can benefit our apporach and can be used to provide us with a protected area of storing the secret keys that are used to encrypt the full application's data.

Table 4.2: Comparison of main memory encryption strategies

Technique	Level	DMA attacks	Cold-boot attacks	Commodity hardware	Runtime Overhead	Platform
Compiler Encryption based [130]	Compiler	No	Yes	No	High	x86
CaSE [206]	Hardware	Yes	Yes	Yes	Medium	ARM
Sentry [50]	Hardware	Yes	Yes	Yes	Medium	ARM
AESSE [128]	Software	No	Yes	Yes	High	x86
TRESOR [129]	Kernel	No	Yes	Yes	High	x86
Loop-Amnesia [170]	Kernel	No	Yes	Yes	High	Generic
vCage [153]	Hypervisor	Yes	Yes	Yes	Low	x86
BitArmor [124]	Physical	No	Partial	No	N/A	Generic
AEGIS [183]	Hardware	Yes	Yes	No	Medium	N/A
This work	Software	Yes	Yes	Yes	High	Generic

4.6 Overview & Limitations

In this work we designed main memory encryption for running processes and we set out to explore the imposed overhead when following different strategies (full Vs. selective memory encryption - dynamic instrumentation Vs. static patching). Contrary to related approaches, our work can be directly applied to commodity systems with AES-NI and IOMMU. Our performance analysis uses both benchmarks and real world applications. The evaluation shows that the average overhead of the encryption cost in real-world applications was 17% and 27% for HTTP and HTTPS respectively.

A major limitation of memory encryption approaches arises in cases where shared memory is deployed across different processes. To communicate correctly, processes have to maintain the same secret key, or use a different secret key which will be used separately for encrypting and decrypting the contents of the shared memory. To deal with such cases, the OS kernel should be responsible for creating different secret keys for each memory segment that is instantiated and attach it to each participant process. Similar inconveniences also arise for devices that allow data transfers via DMA. The exchanged data have to be unencrypted, since the connected devices are not aware of the encryption scheme. As it is easy to overcome these scenarios in hardware-based implementations (e.g. by performing the corresponding cipher operations at the I/O bus), it is not straightforward to provide a solution in software-only approaches. In some cases, where the device already provides a programmable interface (e.g. Endage DAG network cards, general-purpose graphics cards, etc.), it would be possible to implement the encryption and decryption operations on the device and pre-share the secret key.

At this point we should note that after our approach was published, the two major CPU providers (Intel and AMD), presented their own architectural extensions for memory encryption in order to address the attacks we presented in this work. AMD proposed Secure Encrypted Virtualization (SEV) [15] in 2018 which supports running encrypted virtual machines. Each encrypted VM is associated with a unique encryption key in order to prevent other guest VMs and attackers with physical access from accessing its data. Intel, proposed in 2019 Total Memory Encryption (TME) [100]. TME encrypts a computer's entire memory with a single transient key. All memory data passing to and from the CPU is encrypted in order to prevent physical attacks from disclosing sensitive data. We provide more details of these mechanisms in chapter 3. The introduction of these architectural extensions highlights the importance of the work presented in this chapter. Furthermore, these extensions are an even more hardware accelerated version of our work and thus demonstrates the thesis statement of this dissertation regarding the advantages of hardware-assisted security.

Chapter 5
Third party binary library isolation.

In the previous chapter we deployed hardware acceleration for advanced encryption standard (AES-NI) as well as IOMMU in order to prevent physical attacks which aim to steal sensitive data from a system. While, physical attacks are a major threat for modern systems, it is not the only avenue for exploition, since most of today's systems are connected to a network. Thus, attackers are able to launch powerful attacks remotely. This can be accomplished by exploiting vulnerabilities present in modern software, allowing an attacker to access sensitive data or even take full control of the system.

In this chapter, we aim to prevent software exploitation by isolating the execution of possibly vulnerable code that developers include in their applications. In this work, we used a JavaScript runtime environment as a use-case to apply our design. Modern software development relies heavily on third-party libraries. The heavy use of libraries is particularly common in JavaScript applications [113, 135], and especially in those running on the Node.js platform [207], where developers have easy access to hundreds of thousands of libraries through the node package manager (npm). The vast majority of the libraries imported in a Node.js application are implemented in JavaScript and thus enjoy the memory safety guarantee provided by a high-level programming language and enforced by its runtime environment—at times, augmented with language-based protection techniques [14, 62, 121, 188, 193, 194].

Often, however, Node.js applications also import a few libraries written in low-level languages or provided only in binary form. These libraries, termed *native add-ons*, implement either functionality not available yet in the pure-JavaScript ecosystem or components that need to be in low-level languages for performance and compatibility reasons. Native add-ons interact with the rest of the program through a thin JavaScript layer wrapping the library enough to expose Node-specific naming and calling conventions.

Unfortunately, native add-ons are particularly dangerous to the rest of the application—for example, over 20 CVEs are reported for a single string-interpolation add-on [178]. The complete lack of memory safety means that even a single line of memory-unsafe code may compromise the application's safety and security—that is, *including* those of the (safe and

secure) majority of the codebase. Native add-ons can additionally bypass the security guarantees provided by the aforementioned language-based hardening and protection techniques. The exploitation risks of native add-ons compound, as these components are more likely to be targeted by adversaries—exactly because of their vastly higher insecurity and potential impact.

In this work, we developed BINWRAP a hybrid language-binary framework for protecting against the few native add-ons present in modern Node.js applications. We developed a fine-grained read-write permission model applied at the boundaries of native add-ons, offering a unified view and isolation of privilege cutting across the barrier between the language wrapper and the binary core. To achieve this we leverage the protection keys hardware feature, in order to isolate the execution of un-trusted libraries.

Two enforcement components enforce these permissions across language-binary barrier during the execution of the program, protecting both sides of a native addon:

- Language-level interposition protects against unauthorized use of the language-level bindings, and
- Binary-level indirection wraps the entire library and checks permissions to outside interfaces.

The evaluation of our framework proves that BINWRAP can effectively protect *real-world* applications when real-world vulnerabilities are exploited within the loaded native modules. BINWRAP is an efficient solution, imposing 3.17% performance impact on average. BINWRAP is a scalable and practical solution since our design choices were directly influenced by the NPM ecosystem `norms`.

5.1 Background

This section presents background information on Node.js runtime, V8 JavaScript engine (§5.1.1), on restricting memory accesses (§5.1.2), on code reuse attack prevention (§5.1.3), and on system call restrictions (§5.1.4).

5.1.1 Node.js, V8, and NAN

Node.js is a runtime environment for executing JS code, outside the context of a web browser, which primarily targets server-side applications [142]. Internally, Node.js leverages the V8 JS engine [81]. V8 parses JS code and converts that to an *AST* (abstract syntax tree), which can later be "lowered" to V8-specific *bytecode* that, in turn, can be interpreted with the Ignition interpreter [79]; moreover, JS code (in AST or bytecode form) can be compiled to *machine code* using the (optimizing) TurboFan or (non-optimizing) SparkPlug compiler [80].

V8 follows a hierarchical (virtual) memory organization scheme that is primarily geared towards *garbage collection* (GC). Irrespective of how JS code is executed atop V8 (interpreted vs. compiled), JS programs are represented by a so-called *resident set*, which is the collection of memory pages that V8 allocates to facilitate the execution of the respective program. The resident set is further divided to memory (sub)regions that correspond to the runtime (execution) *stack* of the JS program, as well as the *heap*. The latter is designed with aggressive GC in mind, and, to this end, is partitioned to multiple *(semi-)spaces*, which are object allocation arenas that host short- and long-lived objects, in a way that makes GC performant and effective [76].

The heap region also includes special *SLAB-like* [36] areas (or spaces), to support the fast allocation of special, typed objects, "large" (`mmap`-ed) objects, as well as jitted code [77]. Lastly, V8 uses *pointer tagging* to differentiate between plain data and pointer values, while (dynamically-allocated) JS objects are represented by opaque *handles* and accessed/-modified via specific *accessor* functions.

Node.js is built around V8 (both are implemented in C/C++) and provides a rich set of APIs (i.e., the Node-API [137], but also, among others, the V8 API [82]) to JS applications. Most importantly, however, Node.js allows JS programs to load native *add-ons* (modules written in C, C++, ASM, *etc.*), Typically, JS applications leverage add-ons to: (1) perform compute-intensive tasks using highly-optimized C, C++, or even handwritten-ASM code [107]; (2) have access to other (dynamically-loaded) system libraries [152]; (3) interact freely with the underlying OS kernel via the system call (syscall) interface, and utilize system services for which JS abstractions are not available [138]; or even (4) perform computations on specialized hardware, like GPUs and TPUs [103].

Add-ons may *export* functions, and objects, to JS code, directly *invoke* JS functions passed as callbacks, and even *wrap* C++ objects/classes in a way that enables their instantiation directly from JS code (e.g., using the `new` operator). The interoperability between JS and native, C/C++ code is directly facilitated by V8's native code bindings. More specifically, by leveraging any Node API, or even the more esoteric V8 API [82], add-ons can (un)marshal function arguments and return values to/from JS code, invoke JS code (mostly asynchronously), raise (and handle) exceptions, access/pass objects in JS scope, perform JS-to-C/C++ type conversion, and more [137]. Another API designed for Node.js has started providing the NAN (Native Abstractions for Node.js) API [136] as a *portable*, stable API for add-on development, given that both the Node-API and V8 API are version- and platform-dependent (and therefore hinder the portability on add-on code).

5.1.2 Restricting memory accesses

We restrict the memory view of the native module code by leveraging Intel's Memory Protection Keys (MPK) [99]. BINWRAP offloads the execution of native functions to a different

thread. By leveraging Intel MPK we are able to restrict the memory access rights of the thread responsible for executing native functions. Thus, the exploitation of a memory vulnerability in the native module will not affect the rest of the application.

Intel MPK

Intel Memory Protection Keys offer the ability to userspace processes to change access permissions on groups of pages. Each page group is associated with a unique key. An application can have up to 16-page groups. The access rights for each page group are mapped in a thread-local and user-accessible register called protection keys rights register (for user) `%pkru`. Since `%pkru` register is thread-specific, MPK supports per thread view of the process's memory. For example, different application threads have different access rights configured for each key in their `%pkru` register.

The key benefits of MPK over page table permissions are performance and the ability to configure different memory permissions to each thread running in the process. The access rights supported by the page groups are read, write, read-only, and no access. Data accesses on memory pages associated with protection keys are checked both against the access rights defined in the `%pkru` register, as well as the permissions in the page table. Instruction fetching is checked through the permissions in the page table.

If a memory page is executable in the page table but configured with `no access` in `%pkru`, the memory page is treated as execute-only. This occurs since any data access will result in a mismatch between the rights defined in the page table and the `%pkru` register. Linux support execute only memory pages by leveraging MPK. A call to `mprotect` with only `PROT_EXEC` specified as permissions will result in the allocation of a protection key which will be associated with the memory pages passed to `mprotect`. Next, the `%pkru` register will be set to `DISABLE_ACCESS` for the newly allocated protection key, while the page table rights will be set to executable and readable. Any access to execute only pages except for instruction fetching will result in a memory violation exception.

For associating a memory page (or range of memory pages) with a protection key, the Linux kernel implements the `pkey_mprotect` system call. In a similar manner as the traditional `mprotect` system call, it will also set the access rights passed as an argument in the `%pkru` register. The access rights in the `%pkru` register can be modified with the `wrpkru` x86 instruction. Since `%pkru` register is user-accessible, modifying the access rights does not impose significant latency, and it is much faster than invoking memory management system calls (e.g., `mprotect`). Finally, `rdpkru` instructions returns the contents of the executing thread's `%pkru` register.

5.1.3 Code reuse attack prevention

Native modules need to execute Node.js and V8 API functions for benign reasons (e.g., JavaScript object allocation). During API invocations, we need to re-enable memory accesses in the API's function prologue and disable them upon return. Since we can operate on source code level, we ensure that there no occurrences of instructions that can reinstate memory accesses in the native module. However, an attacker could launch a Code Reuse Attack (CRA) that first re-enables memory access and then copies data to an accessible area. To prevent these attacks, we again utilize MPK and also rely on Address Space Randomization Layout (ASLR) to `hide` the location of the trusted code (Node.js) from the untrusted part (native module). With this technique native modules can call Node API functions without ever knowing their real address. In BINWRAP$_B$ we link the native module with a wrapper library that interposes each API function required by the native module. The interposition functions call the actual API functions; however, the memory region they reside is configured as execute-only and thus, the actual address of the API functions is not visible from the native module thread.

5.1.4 System call restrictions

Native modules also execute several system calls for benign reasons. An attacker can misuse these system calls to bypass BINWRAP$_B$ sandbox. In BINWRAP$_B$, we deploy two techniques to restrict system call execution. The first technique relies on finding the actual set of system calls required for the execution of the native module. To extract this set, we deploy Sysfilter [63]. Sysfilter is a static binary analysis framework that extracts an application's set of system calls. The enforcement submodule of Sysfilter produces a BPF filter that can be used with seccomp-bpf [118] to deny the execution of system calls not present in the set.

SecComp

SecComp is a kernel mechanism that can restrict the system calls an application can execute. Since version 3.5 Linux kernel supports SECure COMPuting with Berkeley Packet Filter (seccomp-BPF). The filter rules, allow or deny system calls based on system call numbers and arguments. The applied filter can only be replaced by a more restrictive filter and cannot be removed. The filters applied are per-thread, and thus we can restrict system calls only to the native module thread. However, seccomb-bpf cannot dereference pointer arguments.

Our second technique aims to restrict system calls that attackers can misuse and are present in the system call set required by the native module. For these cases we remove any implicit and explicit `syscall` and `sysenter` instructions from the native module code

and we only allow system calls through `libc` library. Finally, we interpose `libc` functions that wrap system calls required by native modules in order to filter their arguments.

5.2 BINWRAP Overview

We use an image processing library (§5.2.1) to illustrate the problem of a Node.js "module" containing vulnerabilities both on the JS and the native (i.e., add-on) part(s) of its code, and then outline how BINWRAP addresses the respective problems (§5.2.2).

5.2.1 `png-img`: A Node.js Portable Network Graphics Library

Consider a Node.js application that creates PNG image objects from a supplied input *buffer*. More specifically, the developer provides a buffer to a Node.js library (`png-img`), implemented (partially) in native, add-on code, which contains raw image data. In addition, assume that the *size* of the input data is not checked to ensure they fit into the buffer in question, and hence a *memory error* can occur (e.g., a "buffer overflow").

Such errors are a common attack vector when code written in memory- and type-unsafe languages, like C, C++, Objective-C, and assembly (ASM) [195], is involved, and they typically manifest by exploiting missing sanitization logic, pointer arithmetic bugs, invalid type casts, *etc.*—i.e., bugs in code that trigger *spatial* [131] or *temporal* [132] memory safety violations, enabling attackers to *corrupt* or *leak* contents inside the (virtual) address space of victim programs. The code snippet below corresponds to the relevant application fragment of our example.

```
1 const fs = require('fs');
2 const PngImg = require('png-img');
3 let buf = fs.readFileSync('./img.png');
4 let img = new PngImg(buf);
```

First, the developer loads the library `png-img` (ln. 2) to add image processing capabilities in their application. Consequently, they load raw (image) data into `buf`, using the `fs` module (ln. 4). Finally, the `buf` object is passed to the `PngImg` constructor for generating `img`, i.e., the PNG image object.

```
1 const PngImgImpl =
2     require('./build/Release/png_img').PngImg;
3
4 module.exports = class PngImg {
5     constructor(rawImg) {
6         this.img_ = new PngImgImpl(rawImg);
7     }
8 }
```

In the snippet above, we zoom into the step(s) performed by `png-img`, after the developer imports the library to the application. `png-img` uses NAN to link a native function

(written in C, C++, *etc.*) with the `PngImgImpl` object (ln. 1–2). Every time the `png-img` constructor in invoked, the buffer object, which contains the raw (image) data, is passed to the native function (ln. 4–7).

Since the add-on is written in a memory- and type-unsafe language, it may contain bugs (e.g., a buffer overflow, ln. 6) that trigger memory errors [195]. More importantly, given that the raw image data are of *unknown provenance*, attackers may provide specially-crafted inputs that *exploit* the underlying memory errors, potentially resulting in *arbitrary memory read* (disclosure, leak) and *arbitrary memory write* ("write-what-where") primitives [154].

In real-world settings, attackers primarily aim for tampering-with *control data* (e.g., return addresses, function pointers, dynamic dispatch tables) [112], as these facilitate *hijacking the control flow* of the program and performing *arbitrary code execution* [141]—typically via means of *code reuse* [40]: i.e., the attacker executes benign program code, in an "out-of-context" manner, by tampering-with control data; a wide range of code-reuse attack techniques has been proposed and developed thus far [35,44], enabling access control and policy enforcement bypasses, privilege elevation, and sensitive data leakage [185].

Considering these facts, we found a relevant vulnerability of png-img documented in National Vulnerability Database [126].

```
1
2 void PngImg::InitStorage_() {
3     rowPtrs_.resize(.height, nullptr);
4     data_ = new png_byte[.height * .rowbytes];
5
6     for(size_t i = 0; i < .height; ++i) {
7         rowPtrs_[i] = data_ + i * .rowbytes;
8     }
9 }
```

The vulnerability here is that height and rowbytes variables are 32-bit integers and thus can be overflowed. An attacker could trigger this overflow in order to cause an inadequately sized memory allocation. Subsequently, image data will overwrite adjacent memory regions. As we discuss in 5.6, this arbitrary memory write can be leveraged by an attacker in order to take over the control of the application.

5.2.2 Node.js Module Confinement with BINWRAP

To harden the respective Node.js application against vulnerabilities in `png-img`, we apply BINWRAP both at the JS and the native part(s) of the library. More specifically, BINWRAP comes bundled with a set of tools for performing static and dynamic analyses, and policy enforcement, at the level of native, binary code (BINWRAP$_B$), as well on JS code (BINWRAP$_L$). (The latter typically wraps the add-on code and provides a high-level API

for interfacing with Node.js-based application code.) We envision BINWRAP$_B$ as the set of binary-focused methods and techniques that is to be applied during library installation time, whereas BINWRAP$_L$ is the {load, run}-time counterpart, targeting the JS wrapper code.

BINWRAP$_B$

BINWRAP$_B$ consists of a set of memory isolation and code confinement techniques, tailored to the runtime environment of Node.js, which aim at restringing the execution, and the side effects, of unsafe add-on code in *part(s)* of the corresponding virtual address space (VAS). More specifically, the execution of native add-on code is dispatched to a special (Node.js) execution thread, which has a *restricted* memory view of the virtual address space, by leveraging Intel's MPK (Memory Protection Keys) technology [99]. The benefits of this *intra-VAS isolation* are twofold: first, memory errors in `png-img`'s native code *cannot* be used to tamper-with the data of the Node.js runtime —i.e., BINWRAP$_B$ provides data confidentiality/integrity against (arbitrary) memory disclosure/corruption vulnerabilities in unsafe library code; and, second, any potential reuse of code is *limited* to re-using functionality that exists in `png-img` only—i.e., BINWRAP$_B$ prevents code-reuse-based, control-flow hijacking attacks, which originate from the native library, from re-using functionality that exists in the code of Node.js itself or that of any other library in the same VAS.

In addition to the above, a `seccomp-BPF` filter is installed in the special execution thread to further *restrict* the interactions of the latter with the OS, in case the control-flow of the native (library) code is tampered-with (despite being sandboxed). BINWRAP$_B$ automatically extracts the set of system calls (syscalls) required by the native code, and complements that set with syscalls that may result from the invocation of Node.js functionality via NAN (i.e., the native code invokes Node.js code via the NAN API), as well as the invocation of V8 APIs, or `libc` (and other system libraries) APIs.

At runtime, if JS code needs to invoke a native function that belongs to `png-img`, via NAN, BINWRAP$_B$ dispatches the execution of that function to the special thread, which executes the unsafe code under a restricted memory view that is HW-enforced by Intel MPK. Note that the unsafe code may in turn invoke APIs that belong to Node.js, V8, `libc`, or any other system library. In such cases, the control flows to the target (API) entry points (and back) via special *gateways*, which alter the memory view(s) of the code accordingly.

The required analyses for all the above (i.e., gateway generation, syscall extraction) are performed *statically* during the installation of a Node.js library/module that contains native code, and need only to be repeated if the respective code is updated. Lastly, our techniques are *complete* and have minimal requirements (i.e., access to symbols) for increased precision.

BINWRAP$_L$

BINWRAP$_L$ consists of both a static and a dynamic part. By running the static analyzer on the JS wrapper code of `png-img` (i.e., `index.js`), at load-time, we get the following (RWX-like) JSON report that summarizes the developer-intended access permissions regarding the various JS objects involved.

```
1 "/node_modules/png_img/index.js": {
2 "module": "r",
3 "module.exports": "w",
4 "require": "rx",
5 "require('./build/Release/png_img')": "ir",
6 "require('./build/Release/png_img').PngImg": "rx"
7 }
```

Armed with the above access map, BINWRAP$_L$ traces object accesses at runtime and blocks any attempt to access an object in a way that is not compatible with the extracted policy, thereby further *locking* the interaction of `png-img` with the Node.js application that uses it.

5.3 Threat Model

Our threat model assumes that the adversarial capabilities allow the exploitation of memory bugs in *benign* native modules. An attacker therefore, can leverage memory corruption vulnerabilities in order to develop arbitrary memory read and write primitives. The exploitation of these vulnerabilities can be used in order to access sensitive data and even perform Code Reuse Attacks. We do not consider malicious native modules that will actively try to evade our hardening mechanisms. Moreover, we assume that the high-level language part of the library is restricted through existing languages based mechanisms e.g., MIR. Node.js runtime, as well as system libraries are considered trusted and free of vulnerabilities. Finally, we consider side-channel attacks and hardware faults as out of scope.

In order for BINWRAP to protect Node.js runtime environment from these cases, the following OS and hardware features are required. The OS must include Seccomp BPF in order to enable system call filtering. We also consider that W$\hat{\text{X}}$ policy is enforced and that the native module does not include self-modifying code. Moreover, Node.js, system libraries and the native module leverage Address Space Layout Randomization (ASLR). Our framework does not interfere with any other possibly deployed security mechanisms e.g., AppArmor, RELRO, *etc.*. Rather, thse mechanisms can further enhance the protection offered by BINWRAP. The hardware must include Memory Protection Keys or a mechanism that offers equivalent capabilities. While our required hardware feature cannot be con-

sidered as standard as our OS prerequisites, it is part of latest Intel's server CPU series. Moreover, MPK functionality can be emulated through memory tagging which is widely available in ARMs latest processor series [18].

Our techniques aim to address three challenges:

- Prevent the native module thread from accessing memory outside of the native module's loaded address range and its heap-allocated memory.
- Prevent the native module thread from executing CRA gadgets outside of the native module's `.text` area.
- Prevent the native thread from misusing system calls.

We consider the Node.js JavaScript runtime environment and our customized native module layer (NAN) as the trusted part of the application. We consider the source code of the native module as the untrusted part of the application. We deploy many techniques to ensure that any exploitation attempts targeting the native module code will be confined within its bounds and will not affect the whole application.

5.4 Design

The key idea behind the design of BINWRAP is to separate the runtime execution of the untrusted component from the rest of the application. Runtime separation is achieved using different execution threads for the two trust domains, while isolating the thread responsible for executing the untrusted component—effectively, limiting its memory visibility and system-call execution capabilities. BINWRAP limits the memory visibility of the untrusted component by creating a dedicated memory view for the untrusted thread. Also it limits access to the system calls available to the untrusted component, by wrapping and filtering system calls in the untrusted thread.

This section describes BINWRAP's mechanism for isolating the execution of untrusted native components. Fig. 5.1 presents BINWRAP's wrapping. BINWRAP first compiles the source code of the untrusted component and then statically analyzes the resulting binary —a `.so` shared object. We prefer analyzing over the source code since we can find the complete set of external symbols required, for instance calling `printf` will also execute other `libc` functions e.g.,write, *etc.* This analysis aims at extracting (1) the full set of system calls necessary for the execution of the native component, and (2) the set of Node.js-internal API calls used by the native component, e.g., `v8::External::New(v8::Isolate*, void*)`, `v8::Object::Set-InternalField(int, v8::Local<v8::Value>)`, *etc.*.

BINWRAP then creates a custom instance of a Node.js API *layering* library—loaded during the initialization of the native component. This BINWRAP-infused library sets up ap-

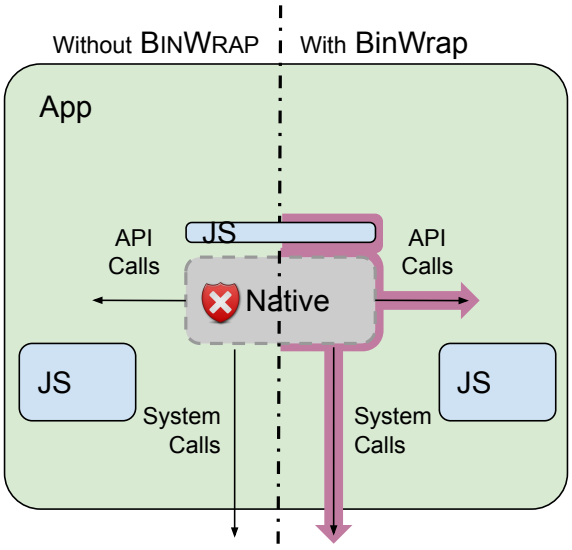


Figure 5.1: Application of BINWRAP framework in Node.js environment.

propriate seccomp filters for the set of system calls extracted in the previous step. BINWRAP then recompiles the native component, linking against the library instance, thus injecting the filter into the native component.

5.4.1 Isolation techniques

Native function execution

Native modules utilize the Native Abstractions for Node (NAN) package to wrap native functions. Native functions are invoked through callback info objects. In BINWRAP we handle these callback info objects to the restricted thread. This thread is initialized during the first time a native function is executed. The Node.js process thread that dispatched the callback info object to the native function thread will block until the native function returns. After the native function returns, the main thread will be unblocked. We used a shared variable as the synchronisation primitive between the two threads. The separate threads design enables BINWRAP to leverage thread specific mechanisms (i.e., MPK and seccomp) in order to isolate the execution of native libraries.

Data access filtering

BINWRAP restricts the memory accesses of third party libraries in Node.js. Since we decouple the execution of untrusted code by creating a new thread, we can prohibit arbitrary accesses to sensitive data stored in Node's memory by leveraging Intel's Memory Protection Keys (MPK) technology [99]. During the initialization of the native module thread,

we associate a protection key with the pages that may contain sensitive data. These data include all the memory allocated and managed by the V8 JavaScript engine. The native module thread will initially change the rights associated to `no access` on the protection key associated with Node's allocated pages. The native module address ranges and allocated memory is excluded by this set. Subsequent memory allocations for expanding V8 memory pool are also associated with Node's protection key. Finally, we ensured that there is no explicit data sharing between Node and native modules (e.g., globals) by analyzing the symbol table of the top 500 popular native module.

An obvious limitation of protection keys is that only 16 are available and thus only 16 different *views* on memory can be supported. This issue has been addressed by the literature through virtualising memory protection keys. In libmpk [146] the authors implement a software abstraction for MPK that virtualises the hardware protection keys. Another solution is grouping sets of native modules under the same protection key. In this approach however, a vulnerable module could also affect the other modules in the set.

Node.js and V8 export a large API set of functions in order to allow native modules to perform various tasks (e.g., object allocation and management, type conversions, *etc.*). Since Node.js is part of the trusted domain when the native thread executes Node API functions, the access rights on Node's data should be re-enabled. In our framework, we modified the API functions to change the rights for the protection key to allow memory operations. We reimpose restrictions before an API function returns back to the native module. During every API call we store the previous contents of `pkru` register in API's function stack variable. During returns, the `pkru` will be modified only if the previous `rights` stored in the stack restrict memory accesses (execution returns to native module). This design choice stems from the fact that API calls may be nested.

An attacker could access sensitive data by harvesting stale data in deallocated stack frames after API functions execution. During the execution of the trusted part, the thread can access any data, and Node API functions can copy data in the stack (as function arguments, *etc.*). After the function returns, the residual data can be accessed without restrictions. To prevent this, the native execution thread zeroes out the deallocated stack frames before returning back to the native module, effectively deleting residual data.

Preventing Code Reuse

An attacker could launch a code reuse attack targeting instruction snippets that remove the restrictions i.e., `wrpkru` and fetch data to memory areas accessible by the native module. These instruction sequences are present in the trusted code since the data access restrictions must be lifted during the execution of the trusted part. These instructions can also be implicitly present in the untrusted part since x86 instructions have variable length. Moreover, `xrstor` instruction can be leveraged to restore a crafted register state with mod-

ified `pkru` in order to allow access on restricted memory. In order to prevent an attacker from using the unlocking functions, we again utilize MPK and also rely on Address Space Layout Randomization (ASLR) to `hide` the location of the trusted code (Node.js) from the untrusted part (native module). The key idea of this technique is that the native module can call Node.js API functions without ever knowing their address.

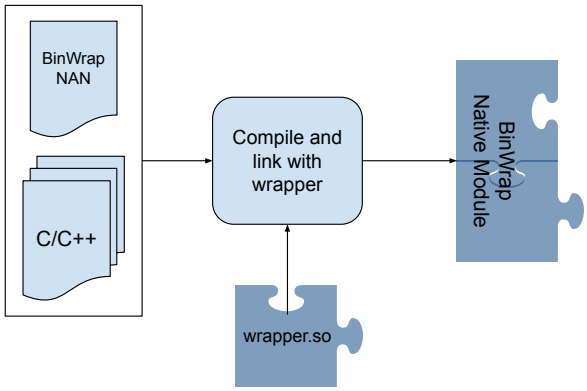


Figure 5.2: Compilation order of BINWRAP native modules.

We designed a custom linking procedure to hide Node API and library locations from the untrusted part. Initially, we extract all the API and library functions needed by the native module from the module's shared object `.plt` section, along with the offset in the `.got` section. Then, we create a new *wrapper* shared object which contains a wrapper function for every Node.js API and library function required by the native module. We link the wrapper library to the native module and manually resolve the native modules dynamic symbols to point at the wrappers (at load time). Next, we utilize the capability from MPK to mark the wrapper functions as execute only. Thus, arbitrary reads will fail to reveal API and library locations finally, ASLR also ensures that the address of Node.js and linked libraries is different on separate executions. Fig. 5.2 presents a high-level overview of BINWRAP modules compilation procedure. Finally, the wrapper library implements dynamic symbol interposition to filter system calls that can bypass the native module sandbox if misused.

To prevent attacks targeting `implicit xrstor` and `wrpkru` instructions, we scan the binary with ROPgadget tool [161]. We vet any occurence of these instructions in a similar manner as G-free [140]. Since we can also operate on the source level of the native module, we do not only rely on Static Binary Instrumentation (SBI) to vet unsafe instructions, rather we can transform the source code to prevent unsafe instructions from being emitted in the final binary. Our analysis on the top 500 native modules with ROPgadget tool found no implicit occurrences of `xrstor` and `wrpkru` instructions. The chances of implicit occurrence are low since both of these instructions are more than 3-bytes. We do not need

to vet unsafe instructions in Node.js or the linked libraries (e.g., libc), since we wrap all the dynamically linked symbols with execute only wrappers in order to hide their actual location in the memory.

5.4.2 System call extraction

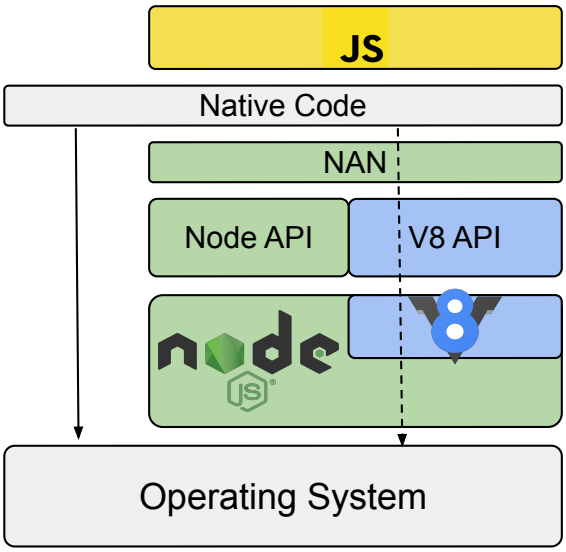


Figure 5.3: Classification of required system-calls.

Native modules often depend on system calls for key functionality available by the operating system. There are two avenues native modules issue system calls. First, system calls are by default available directly to the native component—e.g., component calls `mmap` for mapping memory. Second, they are available indirectly, through Node.js-internal APIs that the component uses—e.g., component calls `v8::External::New(v8::Isolate*, void*)` from Node core, which internally calls `brk` system call. Since native components make extensive use of Node.js-internal APIs, the resulting combined set of system calls used by the native component may be large. Both of these classes can be used as a means for an attacker to cross the protection boundary, effectively bypassing BINWRAP's enforcement mechanism [52, 196].

To extract the full set of system calls a native module requires, we use an intra-procedural, flow-sensitive binary analysis. We analyse each native module in order to extract both the directly included system calls, as well as the system calls inherited through V8 and Node.js API functions (dashed arrow) Fig 5.3.

Direct System Calls

The analysis first receives as input the shared native component, i.e., a `.node` file. It proceeds to resolve dependencies to shared libraries, and then (over-)approximates function-call graph (FCG) of the native component. This approximation is constructed over all objects in the scope of the component and its dependencies. The analysis then performs a set of analyses atop the FCG to extract a tight (but safe) set of developer-intended system calls.

Inherited System Calls

To identify the Node.js-internal API functions used by the native component, we first analyse the native module's shared object symbol table. We then uses each function symbol as entry point for analyzing the Node.js executable and identify the reachable system calls. The analysis trades soundness for completeness, in that system calls performed by the native component will exist in the extracted set—but not all extracted system calls are expected to be used in every (or, indeed, any) execution of the native component. The set of extracted calls includes system calls directly used by the native component, calls used by Node.js-internal APIs during that execution, system calls in `libc`, and any other shared libraries loaded dynamically.

5.4.3 System call restriction

BINWRAP uses the extracted system-call set to create a filter containing the complete set of system calls that may be executed by the native thread. Given a set of allowed system-call numbers, BINWRAP's enforcement tool first converts them to a BPF program and then uses `seccomp-BPF` to execute, and thus enforce, this filter during the execution of the unsafe thread. The core of the BPF filter's logic is centered around conditionals that check the system call number and non-pointer system call arguments. To inject the filter into the binary program, BINWRAP leverages BINWRAP-specific Node.js API templating. BINWRAP provides custom templates of Node.js API libraries that contain placeholder segments instantiated with custom filter instances. Different Node.js API custom wrappers—e.g., NAN, N-API *etc.*—correspond to different templates. On the other hand system calls requiring pointer argument filtering (described in 5.5.3) are interposed in the library wrapper object through their respective `libc` function.

BINWRAP then instantiates each template (still as source code) using (1) information extracted from the earlier static-analysis phase, and (2) additional hardcoded policies for system calls than can be potentially misused. It then compiles the native component, linking against the Node.js API instance, that contains the `seccomp-BPF` filter corresponding to this native component and the Node.js, V8 and dynamic library interposition wrapper

object. Loading the compiled native component at runtime will result in the untrusted thread executing the appropriate `seccomp-BPF` filter upon initialization.

5.5 Implementation

Our framework applies across the whole stack of a Node.js application. The JavaScript code of the third party library is analyzed with BINWRAP$_L$ to extract the permission model that will be enforced at runtime. Our NAN modifications add 200 lines of code for initializing the native execution thread, the synchronization (i.e., the execution of native functions) and the seccomp-bpf configuration. We chose NAN due to BINWRAP$_L$ compatibility, however our implementation is directly applicable to N-API as well as Node.js.

Our wrapper library is generated using bash scripts and ranges between 240 and 600 lines of code depending on how many V8 and Node.js symbols are dynamically linked to the native library.

5.5.1 Node and V8 API modifications

We modified any V8 and Node API function reachable through NAN API. We identified the full set of these functions by analysing the test binaries shipped with NAN package. Our analysis discovered 122 dynamic symbols that point to V8 and Node.js. We additionally analysed the native modules that consist our evaluation set and found that they link less than half of these functions i.e., around 50 symbols. The modifications are functions that remove memory restrictions upon entry and reimpose them before the API function returns to the native module.

5.5.2 Wrapper library

Wrapper libraries are generated using bash scripts and ranges between 240 and 600 lines of code depending on how many V8 and Node.js symbols are dynamically linked to the native library. The wrapper functions are pure (i.e. they do not create a stack frame) and consist of two instructions implementing an indirect jump. `mv` and `jmp`. System calls with pointer arguments that can be potentially misused are intercepted by preloading their `libc` wrapper. The wrapper includes a constructor method that will be the first function executed when the native module is loaded. Each pure wrapper is patched by the constructor in order to store the wrapped symbol's address in the auxiliary register, which will be dereferenced during the indirect jump instruction. The native modules `.got` is configured to point at the wrapper functions. Finally, the constructor maps the wrappers as execute only. This will cause the allocation of a new memory protection key associated with the pages containing the wrapper functions.

```
1 __attribute__ ((aligned(4096), pure))
```

```
2  void
3  wrap_node_api_func(){
4      asm ("movq 0xdeadcafe, %rax; jmpq *%rax");
5  }
6  ...
7  static __attribute__((constructor)) void
8  init_method(void){
9      ...
10     mprotect((void*) wrap_node_api_func, 4096, PROT_WRITE);
11     rewrite_loc = wrap_node_api_func;
12     *rewrite_loc = &node_api_function << 16 | 0xb848;
13     ...
14     mprotect((void*) wrap_node_api_func, 4096, PROT_EXEC);
15     write_got = native_module_address +
16                 node_api_func_got_entry;
17     *write_got = wrap_node_api_func;
18     ...
19 }
```

The constructor method first finds the location of the native module's shared object in the process memory map. Then, the addresses of each symbol are collected using `dlsym`. The wrapper functions load an address in the auxiliary register `%rax` and then indirectly jumps to that address with `jmpq *%rax`. The constructor then marks wrapper functions as writable and patches the `mov` instructions in order to store the actual symbol's address. Then the native module's GOT is patched to point at the wrapper functions. Finally, the wrapper functions are configured as execute only.

5.5.3 System Call policies

Several system calls can be leveraged to bypass MPK based restrictions [164, 196]

Memory management

System calls that are used for memory management, like `mmap`, `mprotect`, `munmap`, `brk`, or `mremap`, could be used to allocate executable memory, execute `pkey_set` instructions or move data to memory locations that can be accessed. Protection keys can also be wiped by de-allocating and re-allocating the target memory region. In BINWRAP, we hook these system calls and disallow them to target the protected domains as well as allocate executable memory. We also disallow remapping executable pages since it is possible to form implicit unsafe instructions on page boundaries. Using `personality` with `READ_IMPLIES_EXEC` an attacker can render any subsequent allocated pages executable. None of the native modules required this system call, and thus we safely deny its execution. We finally disallow `userfaultfd`, since it can enable arbitrary writes on MPK-protected pages [164].

Process and thread control

Another family of dangerous system calls is related to process creation (`fork`, `exec`, and `clone`). These system calls create a new process that is either new (exec system calls), or executed in the same address space. In `clone` the MPK configuration is inherited to the new thread, and is thus safe. `Fork` system call however, can be combined with `kill` system call in order to force the child process to produce a core dump which can be read by the untrusted domain. We found that `fork` and `exec` are not required by native modules and deny their execution. We also disable core dumps by configuring the application process with `(PR_SET_DUMPABLE, SUID_DUMP_DISABLE)`. This kernel utility also prohibits access to `procfs` and thus prevents the misuse of file-related system calls. We also safely deny `prctl` and `set_thread_area` which can remap thread-local storage.

Signal handling

During signal delivery, the kernel stores the register state (including `pkru`) on the stack. When the signal handler finishes its execution, the register state is restored through `rt_sigreturn` system call. An attacker can craft a register state where `pkru` register allows memory access and execute a sigreturn gadget in order to obtain universal access. There is no wrapper for `rt_sigreturn` in `libc`, since it is not supposed to be called by applications. However, an attacker can call `rt_sigreturn` through reusing `syscall, sysenter` instructions in the `.text` area. In BINWRAP, we treat such instructions as unsafe, and we vet them. Thus, there is no way for an attacker to call `rt_sigreturn`.

5.6 Evaluation

To evaluate BINWRAP, we use a set of real-world native npm packages, investigating the following questions:

- **Q1** How effective is BINWRAP at defending against attacks that exploit real-world vulnerabilities? (§5.6.4)
- **Q2** How much BINWRAP reduces the set of system calls in the context of native modules, what is the set breakdown? (§5.6.5)
- **Q3** How efficient and scalable are each of BINWRAP components? (§5.6.6)

5.6.1 Libraries and workloads

We evaluated each of BINWRAP components, testing the security guarantees and the runtime overhead imposed. We investigate each one of the topics, answering a set of relevant questions. To address Q1, we evaluate BINWRAP against exploits targeting vulnerabilities

in popular third-party libraries. The vulnerabilities were pulled from the Snyk [177] vulnerability database, and exploited by us. We found that BINWRAP successfully prevents the exploitation of the selected vulnerable packages. To address Q2 and Q3, we evaluated the overhead of BINWRAP using real world applications that stress individual components and provide insights about the micro and macro aspects of the performance impact. Our results indicate that BINWRAP can offer strong security guarantees to JavaScript applications that utilise third-party native libraries, while imposing an average of 3.17% runtime overhead.

To benchmark each package and application, we tried to execute the test suite that was provided by the developers. If the application developer did not provide any test suite, we used the example code provided in the repository for our evaluation. All of the third-party libraries used in the evaluation ship with npm based test suites. We used two sets of benchmarks during our evaluation. The first and major set consists of benchmarks that implement the behaviour of an actual application that uses the third party library (npm packages and applications), i.e., executing mostly JavaScript code and offloading heavy computations on the native module. The second set consists of benchmarks that execute native functions in tight loops, thus stressing the cross-domain transition of our framework. The first set is suitable for presenting the performance impact of BINWRAP in real-world applications, while the second is suitable for measuring the micro aspects of the runtime overhead.

5.6.2 Setup

Our system is configured with an Intel Core i9-10900 CPU with 32GB RAM and runs Linux version 5.4.0-84. We implemented our modifications on Node version 8.9.4. BINWRAP does not require any kernel modifications to run and only needs MPK and seccomp-bpf to be available on the system. Since we run our benchmarks in the latest Ubuntu distribution, we had to recompile each library that Node.js loads dynamically to remove Intel CET instrumentation, which is added by default in most packages. Intel CET uses a customized `.plt` section that is not supported by sysfilter. Disabling CET does not affect the security of the system, since hardware support is not available in our system's processor and CET instructions are treated as NOPs. Finally, we disabled cstates and Intel turbo boost and locked the clock frequency at 2.8GHz.

5.6.3 Evaluation set

To find a representative set of libraries to evaluate BINWRAP, we analysed the entire NPM ecosystem. The goal of this analysis was to find applications covering the following criteria. The applications have (i) large number of dependents, (ii) compatibility with our tool, and (iii) security exigency. It is also of critical importance that the npm package is

Table 5.1: Third party libraries available in npm and applications using these libraries. * C lines of code. ** Number of System Calls

Name	Description	Dependents	CLoC*	NoSC**	Test Type
`node-sass`	Style sheet preprocessor	8457	37365	93	Macro
`bip32`	Bitcoin wallet client	632	5559	82	Macro
`xml.js`	Xml and Sax parser	344	170K	93	Macro
`iconv`	Text recoding	329	96967	91	Micro
`zeroMQ`	Networking library	323	8131	114	Macro/Micro
`node-ref`	Memory Buffer Utilities	289	6900	91	Macro
`tiny-secp256k1`	Optimised library for ECDA	187	24511	82	Macro
`heap-dump`	V8 heap dump	173	6395	91	Macro
`ttf2woff2`	TTF to WOFF2 converter	155	28185	91	Macro
`pty.js`	Pseudo terminal for Node	128	6999	93	Macro
`blake2`	Hash function library	19	26207	91	Macro
`pngImg`	Png Image processing library	6	66244	93	Macro
`picha`	jpeg Encoder/Decoder	4	7522	92	Macro
`statvfs`	File system information	3	6463	93	Micro
`mtrace`	Native memory tracing and logging	3	6263	91	Micro
`node-uriparser`	Native library for URI parsing	3	8111	90	Macro/Micro
`node-hll-native`	Hyper log log algorithm	2	6663	91	Macro/Micro
`syncrunner`	Return output from binary execution	2	10363	91	Micro
`node-fs-ext`	File system utilities	0	6863	95	Micro
`node-delta`	Delta compression algorithm	0	6680	82	Macro
`Video Thumb Grid`	Video thumb grid generation using `picha`	na	7522	92	Application
`Manta Minnow`	Storage utilization agent for Manta project, uses `statvfs`	na	6463	91	Application

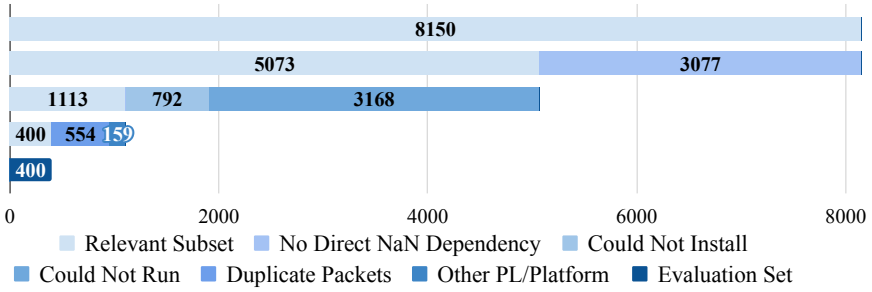


Figure 5.4: Overview of the analysis for the evaluation set.

not corrupted and can run successfully in the base-line case (i.e., unmodified, without BINWRAP).

We took multiple steps to get from the 1,508,366 libraries to the 20 in the evaluation set. First, we found all the libraries that have NAN as a dependency, i.e., 5.073 packages. We only consider packages that can be installed without manual effort (4.201 packages) while we remove duplicates (3.508 packages). Next, we selected packages shipped with test cases that could run out of the box (400 packages). Finally, we reduced our set to 20 based on our criteria.

The evaluation set includes packages that do not have large number of dependents. For example `uriparser` and `node-hll-native` have under 5 dependents. However, these libraries are shipped with tests that are suitable for benchmarking the micro aspects of BINWRAP (i.e., tight oops executing third-party library functions). On the other hand `statvfs` and `picha` are required by `Manta Minnow` and `Video Thump Grid` respectively, which are

Table 5.2: Exploits

CVE	Type	Module	Result
CVE-2018-11499	Use-After-Free	`node-sass`	Information disclosure
CVE-2018-18577	Heap Buffer Overflow	`picha`	Arbitrary Write
CVE-2019-3822	Stack Buffer Overflow	`node-libcurl`	Code-Reuse Attack
CVE-2020-28248	Heap buffer Overflow	`png-img`	Code-Reuse Attack

complete Node.js applications and offer insights with regards to performance in real-world scenarios. With regards to `node-fs-ext, syncrunner, node-delta and mtrace` where chosen to diversify the different types of libraries in our set (i.e., we could choose more popular parser packages, but this would result in a set of libraries with similar behaviour). Finally, `png-img` is also used in our security evaluation, since it contained known vulnerabilities.

5.6.4 Security evaluation (Q1)

To assess the security of BINWRAP we implemented exploits for four distinct CVEs. We analyzed vulnerabilities reported in Snyk.io [177] database for npm packages. These vulnerabilities occur from memory bugs in the shared object that ships with npm packages. To evaluate the security of BINWRAP, we exploited these vulnerabilities and bypass the boundaries of the untrusted part. We present an overview of the exploits in Table 5.2.

```
1 std::string exp_src = exp->to_string (ctx.c_options);
2 Selector_List_Obj sel = Parser::parse_selector
3       (exp_src.c_str(), ctx, traces);
4 parsedSelectors.push_back(sel);
```

CVE-2018-11499 Is an information disclosure attack that exploits a use-after-free vulnerability. The vulnerability was present in `node-sass` package until version 3.5.5. The use-after-free occurs due to lack of exception safety in a loop. In the above code example, `exp_src` is allocated in the stack, while `parse_selector` stores a pointer in a memory buffer. If the parse_selector throws an exception, the stack is unwinded and the `exp_src` buffer is deallocated. During the construction of the exception message, the exception handler de-references the freed memory region. Our exploit manages to leak pointers to heap addresses, which can be used to perform arbitrary reads. With BINWRAP, any attempt to

read beyond the memory allocated for libsass fails due to the restrictions imposed using memory protection keys.

CVE-2018-18577 Is a heap buffer overflow vulnerability enabling arbitrary writes. The exploit is present in `libtiff` that `picha` npm package, loads for processing tagged image file format files. The bug stems from the fact that, `libtiff` ignores the size of the destination buffer when decompressing JBIG compressed images. Thus, an attacker can write arbitrary amounts of decoded data in the destination buffer. With BINWRAP, we are able to prevent this exploit from overwriting data that do not belong to the native module's memory address ranges. In this scenario, an attacker could replace sensitive data used by the JavaScript part of the application. Since these areas are not accessible by the thread executing `libtiff`'s code, when a store instruction targets a memory address beyond its visibility, a memory violation exception is raised.

CVE-2019-3822 Is a stack buffer overflow that can lead to the execution of a ROP gadget chain. The `node-libcurl` npm package links to `libcurl` library, which had a stack buffer overflow vulnerability from version 7.36.0 to 7.64.0. The vulnerability is due to the fact that during an NTLM negotiation, `libcurl` sends a message to the server containing the server's original response. If that response is large enough, it leads to a stack buffer overflow. In our version, the response buffer was statically allocated with 1024 bytes. The response buffer is `base64` decoded and also `memcpy` is used instead of strcpy (making it easier to exploit, since copies will not stop when zero is encountered). We used Ropper [162] to create a ROP chain that loads the command we want to pass to `system libc` function and execute it. This exploit does not work when BINWRAP is deployed, since `system` function will end-up executing `execl` system call. In every native module we analysed with sysfilter, `execl` system call was not present in the system call set that the native module might need to execute (for benign reasons), thus the execution of `execl` fails.

CVE-2020-28248 This vulnerability leads to an under-allocated buffer due to an integer overflow in a memory initialization function. The exploit is present in `png-img` npm package in all versions up to 3.1.0. This condition introduces a heap-based buffer overflow that can be exploited using a specially crafted `png` file.

```
1 void PngImg::InitStorage_() {
2     rowPtrs_.resize(.height, nullptr);
3     data_ = new png_byte[.height * .rowbytes];
4     for(size_t i = 0; i < .height; ++i) {
5         rowPtrs_[i] = data_ + i * .rowbytes;
6     }
7 }
```

The `InitStorage` function height and `rowbytes` are 32-bit integers, thus this calculation can easily overflow and allocate an inadequately sized region. Data from the decoded image may then be written past the end of the buffer. Library `libpng` registers a callback

function in a struct for reporting errors. The inadequately sized buffer is in the lower addressed region of the struct containing the error callback function. This vulnerability can be used to overwrite the callback In our exploit we overwrite the error callback with the address of `system` function. Similar to **CVE-2019-3822** this exploit does not work with BINWRAP deployment, due to system call filtering.

5.6.5 System call set analysis (Q2)

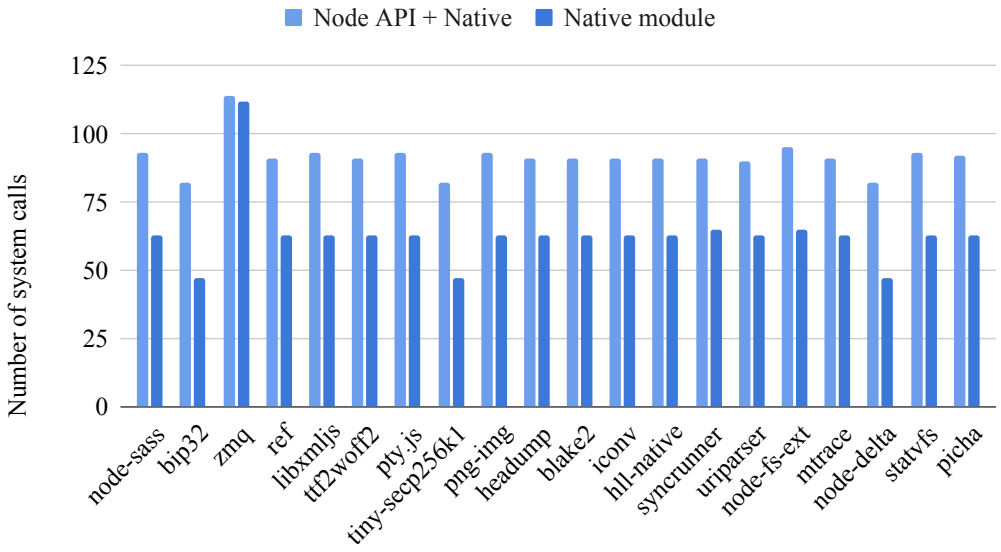


Figure 5.5: System call set size for the native module and the combined with the Node.js API.

We analysed 20 native modules with sysfilter to (i) extract the set of system calls required by the native module and to (ii) extract the system calls required through Node.js API functions. We found that Node API functions require the same 62 system calls in all native modules. In some native modules, like `node-sass` or `zeromq`, the system call sets also contained `inotify_rm_watch` and `epoll_ctl`. We found that the native module shared object requires mostly the same system calls as the Node.js API functions. The number of system calls inherited from Node.js range from 2 (zeroMQ) to 35 (node-delta). As we can see in Fig. 5.5 we can safely block more than 2/3 of the available system calls in most cases, except `zeroMQ` which requires 114 system calls due to handling sockets.

Table 5.3 presents the system calls prone to be misused within the native module for escaping the sandbox. These system calls are hooked within the native module wrapper to filter the arguments passed as discussed in subsection (§5.5.3). Thus, we can prevent

Table 5.3: Subsets of system calls that can be potentially misused to escape BINWRAP.

Type	Subset
Memory Management	`mmap`, `mprotect`, `munmap`, `brk`, `mremap`, `madvise`, `shmget`, `shmat`
Process control	`clone`, `exit`, `kill`, `vfork`,
File related	`read`, `write`, `close`, `pread`, `pwrite`, `readv`, `writev`
Signal handling	`rt_sigaction`, `rt_sigreturn`

the misuse of these system calls while not affecting the correctness of the execution. By deploying Sysfilter we can safely deny 2/3 of the available system calls and significantly reduce the attack surface.

5.6.6 Performance Evaluation (Q3)

In this section, we present the performance impact of BINWRAP when enabled on third party libraries. We compare the runtime performance of BINWRAP against the unmodified version of Node.js runtime environment. We breakdown BINWRAP in 3 different parts to measure the performance overhead. The parts are the (i) native function sandbox, the (iii) dynamic analysis privilege checks, and the (iii) combined impact on performance.

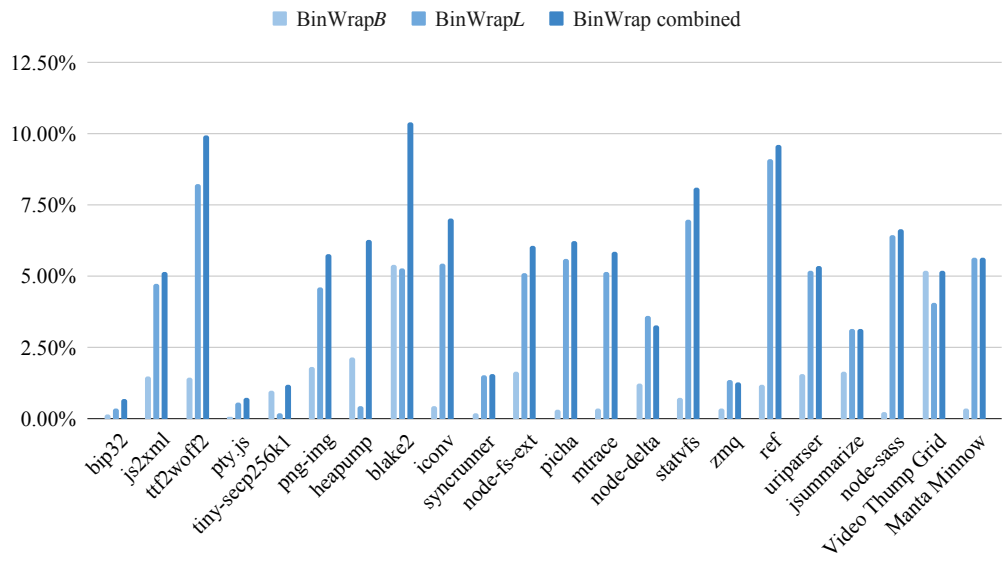


Figure 5.6: Runtime overhead when deploying BINWRAP$_B$, BINWRAP$_L$ and BINWRAP.

Macro benchmarks We evaluated BINWRAP by running the test suite provided by each npm package. We run `npm test` command 100 times and measured the average execu-

tion time for the unmodified npm package and with BINWRAP enabled. The results indicate that BINWRAP imposes minimal overhead. The typical workload is the execution of JavaScript code with sporadic invocation of native functions. The overhead of the dynamic enforcer of BINWRAP is bound to the size of the access rights extracted during the analysis. The overhead originating from the modifications in the native module's shared object is related to the synchronisation between the Node and the native module thread. Since `pkey_set` instructions are executed in user-land, thus changing the rights during domain switches imposes negligible overhead. Interposing system calls and Node.js API functions is also lightweight since the number of extra instructions executed due to interposition is small.

Micro benchmarks During domain transitions, the native module thread is unlocked and executes the native function. The main thread waits for the native module thread to finish the callback execution. We evaluated several synchronisation algorithms to measure the performance in this scenario. Our baseline micro benchmark is a function that increments a global variable, called 100M times. Next, we implemented the same scenario but this time the process spawns a thread that will be responsible for incrementing the variable. The new thread increments the variable once and then locks until the main thread unlocks it. In a similar manner, the main thread will lock until the thread responsible for incrementing the variable unlocks the synchronization variable. When utilising futex's the overhead compared to the benchmark without threads is 240x. When using inline assembly memory operations to spin on the synchronization variable the overhead was reduced to 80x.

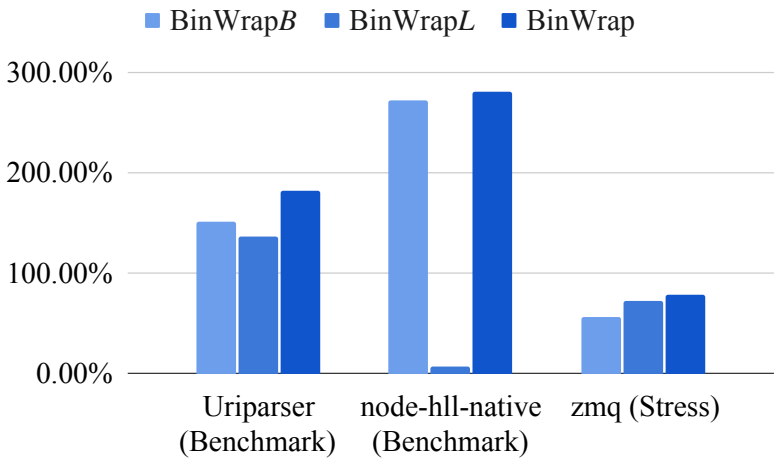


Figure 5.7: Packages that stress the micro aspect of BINWRAP.

We also evaluated BINWRAP with test cases included in NPM packages that stress the

security mechanisms, we present the results in Fig. 5.7. In the case of `uriparser` the benchmark code is only two loops that parse a `URL` 2M times. The first loop uses the JavaScript implementation of the parser, while the latter uses the natively implemented parser. The majority of the code executed triggers the synchronization mechanism between the Node and the native module thread. A similar scenario appears in `node-hll-native`. The benchmark implements a tight loop that executes a native `hyperloglog` function 50M times, which only executes 300 instructions. Finally, `ZeroMQ` benchmark, consists of two instances (sender and receiver) communicating with small (1KB) TCP packets. The receiver expects 1M packets from the sender. In this case, both the synchronization and the system call filtering components are stressed, however a lot of the overhead is amortised due to the execution of network related system calls. On average BINWRAP imposes a performance overhead of 3.17%..

5.7 Related Work

Table 5.4: Comparison of MPK sandboxes. [1] Only addresses system call issues and is based on Donky. [2] Is an API for MPK based sand boxes. [3] PKRU-safe does not address MPK based sandbox issues i.e., system calls, stray unsafe instructions. [4] Cerberus does not prevent exploitation through sigreturn

Technique	In-Process Isolation	No Kernel Modifications	System Call Restriction	Unsafe Instruction Vetting	PKU Pitfalls Protection [52]	New PKU Pitfalls Protection [196]	Performance Overhead
Erim [192]	Yes	No	Partial	Partial	No	No	Low
Hodor [91]	Yes	No	Partial	Partial	No	No	Moderate
Donky [165]	Yes	No	Partial	NA	No	No	Low
Jenny [164]	Yes[1]	No	Complete	Partial	Yes	Yes	Moderate
Cerberus [196]	Yes[2]	No	Partial	Partial	Yes[4]	Yes	Low
PKRU-Safe [108]	Yes	Yes[3]	No	No	No	No	Low
BinWrap	Yes	Yes	Complete	Complete	Yes	Yes	Low

5.7.1 Intra-process isolation

Operating systems focus on process isolation (virtual memory, etc.) to prevent process's from arbitrarily interfering with between them. Intra-process isolation, is required in applications that need to isolate components in the same process. For example, web browsers isolate the execution of different pages in order to prevent malicious pages from accessing sensitive data. A notable family of Intra-process isolation techniques is Software Fault Isolation (SFI), SFI instruments memory operational in order to restrict memory access beyond a designated area. Other instrumentation approaches ensure that out-

of-bound pointers are transformed into in-bound. Research efforts focus on in-process techniques, that offer isolation guarantees with minimum cost [28, 186].

Beyond software-only solutions for intra-process isolation, there are different mechanisms in widely used architectures that can be leveraged for that purpose. BINWRAP utilizes Memory Protection Keys from Intel to differentiate access rights on memory when accessed from the trusted and untrusted parts of the Node.js applications. Several other research efforts, also leverage MPK for intra-process isolation. ERIM [192] and Hodor [91] introduce security domains in applications and protect sensitive data from being accessed by untrusted components. The access rights are modified through call gates (ERIM) and trampolines (Hodor). Moreover, binary inspection is used in order to vet occurrences of MPK instructions. Regarding system calls ERIM only intercepts memory management system calls, while Hodor denies any system call originating from untrusted domains by modifying the operating system. Unfortunately, recent publications [52, 196] present attacks on ERIM and Hodor, extended system call filtering with `ptrace` in ERIM solves some issues but incurs substantial overhead [164].

Donky [165] is implemented on a RISC-V processor, enabling protection keys and user-level interrupts. Domain transitions and memory management system calls are managed by a per-process monitor. Only the monitor has access to the protection key registers. Jenny [164] resolves several limitations of Donky, notably more complete system call filtering. However, Jenny and Donky cannot be directly applied in x86 and require custom hardware.

PKRU-Safe [108] polices inter-domain data flows in MPK based sandboxes. The authors do not address any of the security issues presented in [52, 196] (i.e., system call misuse, stray MPK instructions). Thus, PKRU-Safe is orthogonal to BINWRAP and could be deployed in order to enhance our memory restriction policies (i.e., what can be shared between Node.js and native modules). Cerberus [196] aims to address the issues with MPK-based sandboxes presented in [52] paper and present novel attacks. Cerberus is an API, offering primitives for protecting other MPK sandboxes like Hodor and ERIM. System calls are handled through a kernel-side monitor. However, since the security monitor is implemented in the OS, it is not able to thwart `sigreturn` based attacks.

5.8 Summary

In this chapter we presented BINWRAP: a framework that applies across the whole stack of Node.js applications in order to isolate the execution of potentially vulnerable third party applications. We studied the Node.js ecosystem and understood how native add-ons are used. We leveraged hardware assisted security mechanisms in order to enable strong isolation guarantees. Next, we evaluated the security our framework against real-world exploits in real-world applications. Finally, we evaluated in BINWRAP in terms of performance and

presented an average overhead of 3.17%, proving the statement of this thesis regarding the benefits of hardware approaches in defending modern threat landscapes. We believe that BINWRAP is a practical framework that can protect Node.js applications in the presence of unsafe native libraries.

Chapter 6
Architectural Support for Instruction Set Randomization

In chapters 4 and 5 we presented how architectural features in commodity processors can be leveraged in order to make applications and systems resilient against exploitation. Particularly in chapter 5, we show how we can isolate vulnerable code, written in low-level languages, lacking of memory-safety guarantees. In this chapter, we aim to prevent memory corruption vulnerabilities from escalating to full-fledged arbitrary code execution exploits in non memory safe or type safe applications. We achieve this by designing and implementing architecturally supported Instruction Set Randomization in an open source Sparc V8 processor.

In this work, we extended our previous work [144] with AES and design AESASIST. The AES algorithm, requires significantly more modifications on the processor architecture, however, offers enhanced protection compared to XOR and Transposition. XOR and Transposition require minimal resources in the processor die, however they are vulnerable to key guessing attacks and cryptanalysis.

We compare two techniques for encrypting the executable code: (*i*) statically, by adding a new section in ELF that contains the key that has been used to encrypt all code sections of the binary file, using a binary transformation tool; and (*ii*) dynamically, by generating a random key at load time and encrypting with this key at the page fault handler all the executable memory mapped pages. The dynamic encryption approach supports dynamically linked shared libraries, whereas static encryption requires statically linked binaries. We discuss and evaluate the advantages of each approach in terms of security and performance. Finally, our modified processor can also encrypt the return address at each function call, and decrypt it right before returning to the caller. By doing so, AESASIST is capable to protect against ROP attacks, by (*i*) preventing gadget discovery, via the use of strong encryption schemes (i.e., AES) [39, 169], and by (*ii*) limiting the discovery of call-preceded code locations, via the encryption of the return addresses [42].

To demonstrate the feasibility of our approach we present the prototype implementa-

tion of AESASIST by modifying the Leon3 SPARC V8 processor [4], a 32-bit open-source synthesizable processor [68]. We also modified the Linux kernel 3.8 to support the implemented hardware features for ISR and evaluate our prototype. Our experimental evaluation results show that AESASIST is able to prevent code injection attacks practically without any performance overhead, i.e., less than 1%, when using simple encryption schemes such as XOR and Transposition; more secure ciphers, such as AES, introduce a slightly higher overhead, about 10%, which are acceptable in real scenarios considering the benefits in terms of security. Meanwhile, the hardware extensions add about 10% of additional hardware to support all ISR modes of our design. Our results also indicate that the dynamic code encryption at the page fault handler does not impose significant overhead, due to the low page fault rate for executable pages. This outcome makes our dynamic encryption approach very appealing, as it is able to generate a different, random key at each execution, *transparently* encrypt any executable program, and support shared libraries with negligible overhead.

Overall, the main contributions of this work are:

- We extend our previous work, which was based on simple XOR and Transposition techniques, by adding the AES cipher. By doing so, we can guarantee that an encryption key cannot be derived even if the attacker has access to both the plaintext and the ciphertext (e.g due to a memory leak). In addition, it can hinder any gadget discovery—which actually is a pivotal step of code reuse attacks—that are based in code pointer leaks in order to bypass ASLR and the exploitation of memory disclosure vulnerabilities to map the text segment of a process; in both cases instructions will be encrypted with a strong cryptography scheme that prevents any form of cryptanalysis or bruteforce attacks.

- In order to enable AES, we replicated the cache line fill protocol and placed the ASIST unit between the MMU and the instruction cache. This new MMU component is responsible for decrypting the instructions block-wise before sending them to the cache subsystem. This is a major contribution over the previous design, in which the instructions where decrypted only after they were fetched from the cache, for several reasons: First, it increases the performance due to spatial locality. Second, the new component makes our design ISA-agnostic, hence more easily portable to other architectures, such as CISC.

- We introduce a dynamic code encryption technique that can transparently encrypt pages with executable code at the page fault handler, using a randomly-generated key for each execution. This technique can support shared libraries and does not impose significant overhead.

- In terms of performance evaluation, we performed extra experiments to see how the

instruction cache size affects the performance of ISR, showing that when we increase the cache size (to 32KB or 64KB) the runtime overhead of a multi-round, block, cipher, such as AES is reduced to practical margins. Since we evaluate our design using an FPGA we also offer measurements regarding the area overhead.

6.1 Background

6.1.1 ISR

Instruction set randomization (ISR) has been initially proposed as a countermeasure against code injection attacks [24,104,151]. ISR randomizes the instruction set (ISA) of a processor so that an attacker is not able to know the instruction set of the target machine to inject or disclose code. Therefore, any injected code will fail to accomplish the desirable malicious behavior, probably resulting in illegal instruction execution. To prevent successful machine code injections, ISR techniques typically encrypt the instructions of a program with a specific key. This key actually defines the valid instruction set for this program. The processor decrypts at runtime every instruction of the respective process with the same key. Only the correctly encrypted instructions will lead to the intended code after decryption. Any injected code that is not encrypted with the correct key will result in irrelevant or invalid instructions. At the same time, code fragments, called gadgets, cannot be disclosed in a decrypted form, hence not chained into a meaningful exploit (as it typically happens in code-reuse attacks such as ROP [39] and JOP [34]).

Most of the *early* ISR implementations utilize binary transformation tools in order to encrypt the text section of the target binaries. Running those binaries requires an emulator which is responsible for decrypting the instructions at runtime [37, 104]. Other solutions rely on dynamic binary instrumentation tools [95, 151]. Such approaches have several limitations though: (*i*) They incur a significant runtime performance overhead due to the software emulator or instrumentation tool. This overhead is often prohibitive for the wide adoption of such techniques. (*ii*) Deployment is limited by the necessity of several tools, like emulators, simulators and manual encryption of the programs that are protected with ISR. (*iii*) They are vulnerable to code injection attacks into the underlying emulator or instrumentation tools. More importantly, they do not protect against attacks targeting kernel vulnerabilities [5–7, 45], which are becoming an increasingly attractive target for attackers. (*iv*) Most ISR implementations are vulnerable to evasion attacks aiming to guess the encryption key and bypass ISR protection [144, 179, 200].

6.2 Threat Model

Our threat model includes any programming bugs or inadvertent design flaws in software, that can be exploited by an active adversary in order to launch a code injection or code reuse attack. The adversary does not have administrative access (i.e., superuser) and can have either local or remote access to the system. We also consider a trusted operating system and that the executable files cannot be accessed when stored on disk. The operating system is also protected by our design against code reuse and code injection attacks. Any hardware bugs or vulnerabilities that may arise due to hardware design choices (such as Spectre [109] and Meltdown [119]) are outside of our threat model, as mitigation would require orthogonal solutions for hardware reliability. Finally, we do not target denial-of-service attacks, or side-channel attacks. Below, we list different type of attacks and describe how AESASIST can protect in each case.

6.2.1 In-scope threats

Remote and local machine code injection attacks.

The threat model we address in this work is the remote or local exploitation of any software vulnerability that allows the diversion of the control flow to execute arbitrary, malicious injected code. Moreover, we also prevent the attacks that rely on disclosing the text segment of a process through memory disclosure vulnerabilities, i.e. Code Reuse Attacks. We address vulnerabilities in the stack, heap, or BSS, i.e. any buffer overflow that overwrites the return address, a function pointer, or any control data. We focus on protecting the potentially vulnerable systems against, machine code injection attacks and text segment disclosure.

Kernel vulnerabilities.

Remotely exploitable vulnerabilities on the operating system kernel [5–7,45] are becoming an increasingly attractive target for attackers. Our threat model includes code injection and code reuse attacks based on kernel vulnerabilities. We propose an architecture that is capable of protecting the operating system kernel as well. Our design can also thwart attacks that use a kernel vulnerability to run user-level code with elevated kernel privileges [105].

Return-to-libc and Code Reuse Attacks.

Instead of injecting new code into a vulnerable program, an attacker can execute existing code upon changing the control flow of a vulnerable system to perform the attack. This can be done either by redirecting the execution to existing library functions, attacks typi-

cally known as *return-to-libc* attacks [133]; either by using existing instruction sequences ending with a *ret* instruction (called *gadgets*), a technique known as *return-oriented programming* (ROP) [39, 169]. These types of attacks can be also achieved by chaining gadgets with function pointers residing in the stack [35]. ISR was originally designed to protect a system against code injection attacks, and not to address return-to-libc and ROP attacks [104]. However, ISR with strong encryption can prevent the discovery of usable gadgets, which is the pivotal step of code reuse exploitation techniques. In the case where an attacker has managed to disclose the text pages of a process: the text pages will be encrypted using a different key for each process, prohibiting any gadget extraction (e.g., though cryptanalysis or brute-force attacks). Snow et al. [176] introduced the concept of Just-In-Time-ROP attacks, in which even a single leak of a code pointer and the presence of a memory disclosure vulnerability can be exploited by an attacker in order to map the text segment of a process. This can be achieved by recursively scanning disclosed code pages for more code pointers. A similar technique described by Bittau et al. [31] can circumvent ASLR. Herein, an attacker will exploit available memory disclosure vulnerabilities in order to bypass ASLR. Then, brute force will be used in order to find a ROP sequence that can read the code pointed from the leaked code pointers and send them to the attacker (e.g. write to socket). The attack has been demonstrated to complete within approximately 4,000 requests. Again, ISR makes it hard for an attacker to discover usable gadgets since the text segments will be encrypted.

Key guessing attacks.

Existing ISR implementations are vulnerable to key guessing or key stealing attacks [179, 200]. This way, sophisticated attackers may be able to bypass the ISR protection mechanism, by guessing the key and then injecting and executing code that is correctly encoded with this key. In this work, we aim to design and implement ISR in a way that it will be very difficult for attackers to guess or infer the code randomization key. To achieve this goal, we extend our previous work [144] which was based on simple XOR and transposition techniques, by implementing AES; by doing so, an attacker with knowledge of both the plain-text and cipher-text of instructions will not be able to reconstruct the encryption key.

6.2.2 Out-of-scope threats

Transient execution attacks. Any vulnerabilities in hardware that can disclose protected or inaccessible memory when successfully exploited, cannot be mitigated by AESASIST — by being able to dump the whole memory, an attacker is able to retrieve the process key and circumvent our mechanism. For instance, in the recently discovered Spectre [109] and Meltdown [119] attacks, the instructions that are executed either speculatively or out-

of-order, can cause cache modifications. An attacker that can control the cache and observe the effects of the transiently executed instructions, is capable to disclose otherwise inaccessible memory and retrieve the process key that is used for code encryption. Our mechanism is not designed to protect against such attacks, since they rely on architectural choices rather than software bugs.

6.3 AESASIST Design

AESASIST consists of different modules and components across different layers of the hardware, the operating system, and the user-space, as can be shown in Figure 6.1. Overall, the processor has been extended with two new registers: *usrkey* and *oskey*, which store the keys of the running user-level process and operating system kernel's code respectively. Additionally, a new register *asist_mode* is used in order to select the encryption algorithm. The operating system keeps the key and the mode of each process in a respective field in the process table, and stores the key and the mode of the next process that is scheduled for execution in the *usrkey* and *asist_mode* registers using the *sta* privileged SPARC instruction. Moreover, the processor is modified to decrypt instructions before they are fetched from RAM, using one of the above two keys, according to the supervisor bit.

6.3.1 Encryption

AESASIST offers three different algorithms for code encryption, namely XOR, Transposition, and AES. These algorithms have different performance and security characteristics, as we will see in more detail in Section 6.3.1 and Section 6.5.2, and can be used to cover different needs or requirements. We support two options for encrypting an executable program: static and dynamic. In static encryption, the program is encrypted before each execution with a pre-defined key. In dynamic encryption, a key is randomly generated at the binary loader, and all code pages are encrypted with this key at the page fault handler before they are mapped to the process's address space.

The main advantage of static code encryption is that it has no runtime overhead for XOR and Transposition, while imposing acceptable overhead when using AES. However, this approach has several drawbacks. First, the same key is used for each execution, which makes it susceptible to brute force attacks trying to guess this key. Second, each executable file needs to be encrypted before running. Third, static encryption does not support shared libraries; all programs must be statically linked with all necessary libraries. In contrast, dynamic encryption has a number of advantages: it generates a random key at each execution so it cannot be easily guessed, it encrypts all executables transparently without the need to run an encryption program, and it is able to support shared libraries. The drawback of dynamic encryption is the runtime overhead to encrypt a code page

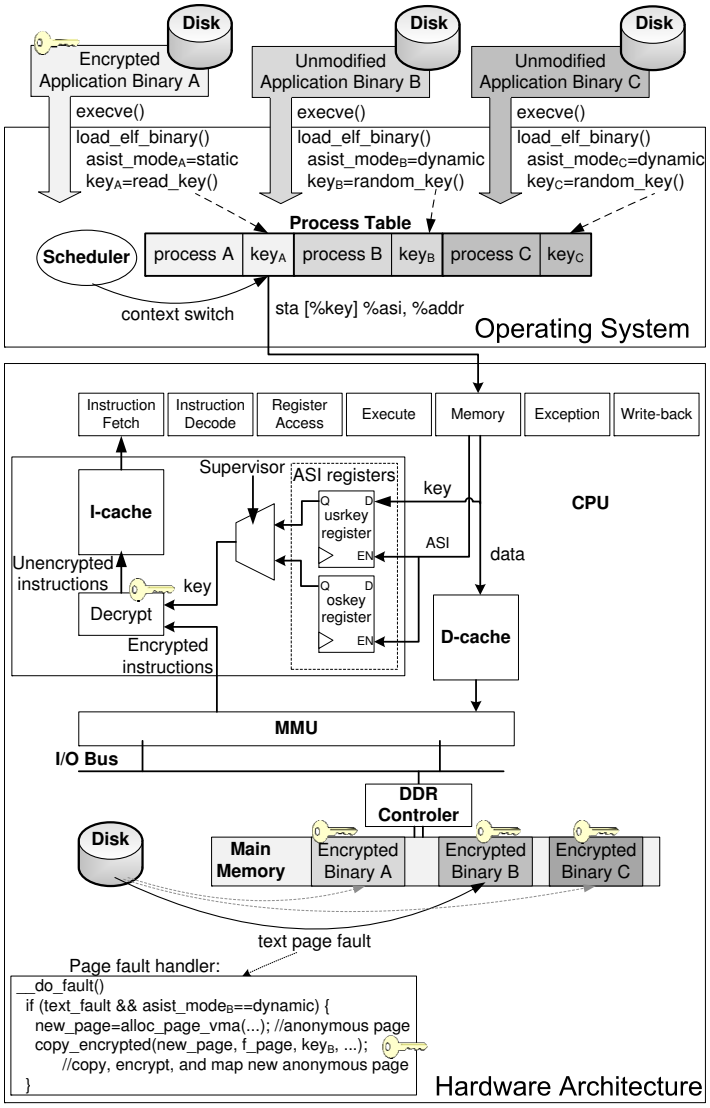


Figure 6.1: AESASISTarchitecture. The operating system reads the key from the ELF binary (static encryption) or randomly generates a new key (dynamic encryption), saves the key in the process table, and stores the key of the running process in the *usrkey* register. The processor decrypts each instruction (or I-cache line in case of AES) using *usrkey* or *oskey* register, according to the *supervisor* bit.

when it is loaded to memory at a code page fault. In Section 6.5 we show that due to the low number of code page faults, dynamic encryption is very efficient.

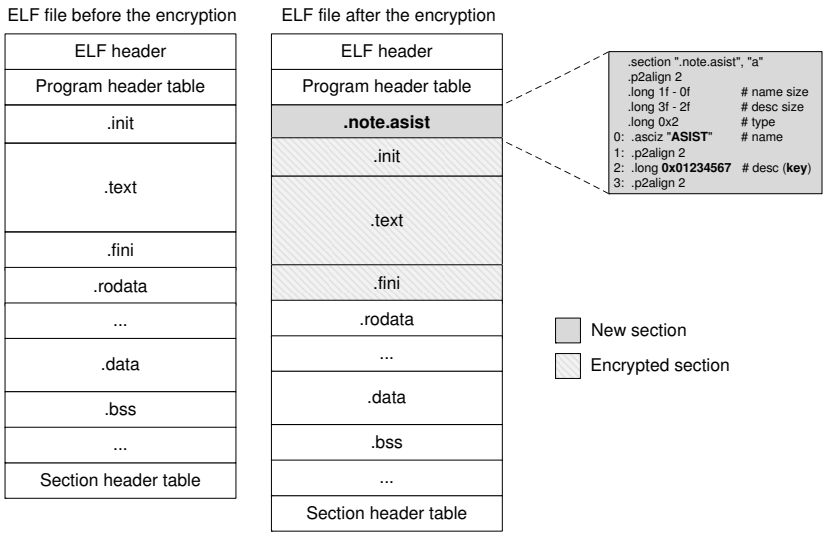


Figure 6.2: The ELF format of a statically encrypted executable file. The key is stored in a new note section inside the ELF file, and all the code sections are encrypted with this key.

Static Binary Encryption

To statically encrypt an ELF executable we extended `objcopy` with a new flag (`---encrypt-code`). The encryption key can be provided by the user or randomly chosen by the tool. Figure 6.2 shows the modifications of a statically encrypted ELF binary. We add a new note section (`.note.asist`) inside the encrypted ELF file that contains the program's encryption key. We also changed the ELF binary loader in the Linux kernel to read the note section from the ELF, get the key, and store it in a new field (`key`) of the current process. In this operation mode we set a new field per process (`asist_mode`) to static. The key is stored in the process table and is used by the kernel to set the *usrkey* hardware register each time this process is scheduled for execution.

Our static encryption tool also finds and encrypts all the code sections in ELF. Therefore, all needed libraries must be statically linked, to be properly encrypted. Moreover, it is important to completely separate code from data into different sections by the linker. This is because the encryption of any data, which are not decrypted by the modified processor, will probably disrupt the program execution. Fortunately, many linkers are configured this way. Similarly, compiler optimizations like *jump tables*, which are used to perform faster switch statements with indirect jumps, should be also moved to a separate, non-code section.

To address the issue of using the same key at all executions, which may facilitate a key guessing attack, one approach could be to re-encrypt the binary after a process crash. An-

other approach could be to encrypt the original binary at the user-level part of `execve()`, by randomly generating a new key and copying the binary into an encrypted one. Even though this approach can occur considerable overheads due to the extra time needed to copy and encrypt the entire binary at load time, we believe that this can be useful to protect against attacks that probe applications continuously in order to find execution traces that do not crash. By re-randomizing the binary code each time, the possibilities to achieve text segment disclosure are diminished.

Dynamic Code Encryption

Our other approach is to dynamically encrypt a program's code before it is loaded into the process's memory. This technique is based on the fact that every page with executable code will be loaded from disk (or buffer cache) to the process's address space through a page fault, the first time it is accessed by the program. By encrypting the code page at this point, AESASIST will dynamically encrypt only the code pages that are actually used by the program at each execution.

To support this, the ELF binary loader is modified to randomly generate a new key, which is stored into the process table. It also sets the `asist_mode` field of the current process to dynamic. The code encryption is performed by the page fault handler at a text page fault, i.e.,, on a page containing executable code, if the process that is responsible for the page fault uses dynamic encryption according to `asist_mode`. Then, a new anonymous page is allocated, and the code page fetched from disk (or buffer cache) is encrypted and copied on this page using the process's encryption key. The new page is finally mapped to the process's address space.

Moreover, we allocate an anonymous page, i.e.,, a page that is not backed by a file, and copy the encrypted code on this page, so that the changes will not be stored at the original binary file. Even though processes that run the same code could share the respective code pages in physical memory, we have a separate copy of each page with executable code for each process, as they have been encrypted with different keys. This may result in a small memory overhead, but it is necessary in order to use a different key per process and achieve better isolation. As shown from recent attacks [31] this approach can prevent probed processes to be forced to use the same key every time they are restarted after being crashed by the attacker.

In practice, the memory allocated for code, accounts only for a small fraction of the total memory. Also, we notice that we can still benefit from buffer cache, as we copy the cached page.

Finally, we also modified the `fork()` system call to randomly generate a new key for the child process. When the modified `fork()` copies the parent process's page table, it omits copying its last layer so that the child's code pages will not be mapped with pages

encrypted with the parent's key. To operate correctly, the dynamic encryption approach requires a separation of code and data per each page. For this, we modify the linker to align the ELF headers, data, and code sections to a new page, by adding the proper padding.

Shared Libraries

One approach to support shared libraries is to bind each page with a separate key, similar to Polyglot [171]. By doing so, shared libraries are supported using a per-page regional key. While this minimizes the memory overhead of having multiple copies of the same page for different processes, it also has several drawbacks; for example, it complicates our hardware, affects the context-switch time, and imposes additional overhead in order to fetch the key for decrypting the instructions.

To overcome this, the code of a shared library in AESASIST is encrypted with each process's key on the respective page fault when loading a page to process's address space, as we explained above. In this way we have a separate copy of each shared library's page for each process. This is necessary in order to use a different key per process, which offers better protection and isolation.

Self-Modifying Code

The design we presented does not support randomized programs with self-modifying code or runtime code generation, i.e.,, programs that modify their code or generate and execute new code. To support such programs, we added a new system call in Linux kernel, namely `asist_encrypt(char *buf, int size)`. This system call encrypts the code of length `size` that exists in the memory region starting from `buf` bytes length, using the current process's key that is stored in process table. We note that the `buf` buffer may still be vulnerable to a code injection attack, e.g.,, due to a buffer overflow vulnerability in the program that may lead to the injection of malicious code into `buf`. Then, this code will be correctly encrypted using `asist_encrypt()` and will be successfully executed. Like previous work supporting ISR with self-modifying code [24], we believe that programs should carefully use the `asist_encrypt()` system call to avoid malicious code injection in `buf`. By doing so, ISR can prevent even from JIT-Spraying attacks [33], where an attacker can overwrite JIT code pages residing in the heap with pages that contain malicious code; similar to code injection attacks, the attacker needs to provide encrypted code with the correct key in order to execute arbitrary code.

Encryption Algorithms and Key Size

The simplest, and probably the fastest, encryption algorithm is to XOR each bit of the code with the respective bit of the key. Since code is much larger than a typical key, the bits of the

key are reused. In our prototype we implemented XOR encryption with key sizes that can range from 32-bit to 128-bit. Even though larger keys typically reduce the probability of key guesses and key extraction attacks, still they can be easily exploited when the attacker has knowledge of both the plain-text and cipher-text (e.g., due to a memory leak) [179,200]. A better solution, with slightly more overhead, is to use the *Transposition* cipher algorithm. In Transposition the bits of a 32-bit word are shuffled using an 160-bit key. For each bit of the encrypted word we choose one of the 32 bits of the original word based on the respective bits of the key. That way, the cipher-text constitutes a permutation of the plain-text. Even though the Transposition cipher provides better security than simple XOR, it is vulnerable to anagramming.

To overcome all previous limitations, we have also implemented the AES algorithm. AES can guarantee that even attackers with the knowledge of both the plain-text and the cipher-text (e.g., due to a memory leak) cannot derive the encryption key. The implementation of a hardware-based AES for ISR though is not a trivial thing. One of the major differences of AES compared to XOR and Transposition, is that it operates on 16-byte blocks. Thus, in order to retrieve the plain-text even of a single instruction, we need to decrypt a 16-byte aligned block. Moreover, our AES decryption unit requires approximately 12 cycles for decrypting each block instead of on-cycle decryption offered by the previous two algorithms. Other challenges are related to the specific architecture of the Leon3 processor that we used for the implementation of AESASIST. For instance, the I-cache communicates with the memory interface using a 32-bit AHB bus, hence the cache lines fill at a rate of one instruction per cycle. Moreover, Leon3 uses a critical word first policy, i.e., during a cache miss the instruction which caused the miss will be fetched first instead of the first instruction of the cache line. Then it will be immediately forwarded to both the integer unit and the I-cache. After that the rest of the cache line will be filled. In order to extend the previous design of AESASIST [144] to support the AES algorithm, we had to make significant modifications. As we will see in Section 6.5.2, even when using a complex, multi-round, symmetric encryption algorithm like AES, the impact on the performance is within acceptable margins when configuring our prototype with a typical I-cache size. This is due to the implementation of a dedicated AES decryption unit placed before the I-cache. Thus, we can benefit from the I-cache locality in order to minimize instruction decryptions.

6.3.2 Hardware Support

Placement of the Decryption Unit

The placement of the decryption unit can add extra cycles on the execution pipeline or even break runtime optimizations added by the processor. To avoid such performance overheads, it is important to place the decryption unit as early as possible. We considered two options for placing the decryption unit: before and after the I-cache.

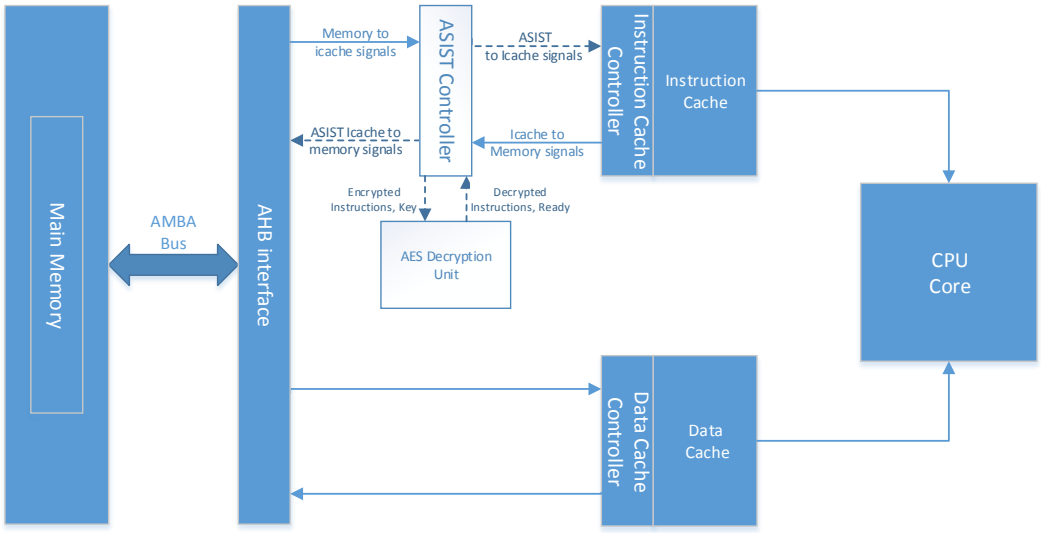


Figure 6.3: The AES extended AESASIST processor is between the AHB interface and the I-cache controller. Each cache block is fetched, decrypted and then sent to the I-cache. To support AES, a separate AES decryption unit is instantiated in order to decrypt each block.

When the decryption unit is after the I-cache, the instructions remain encrypted inside the cache and the decryption takes place every fetch cycle, as shown in Figure 6.4b. This may add extra delays and increased power consumption—especially for complex cipher algorithms such as AES—since it is located in the critical path of the processor. Moreover, since the instructions are encrypted in the I-cache, it may jeopardize any pre-decoding operations located after the instruction cache, such as trace cache [160]. If it is stored encrypted in the I-cache, pre-decoding cannot be performed, which may decrease performance. Some pre-decoding may be quite involving (e.g. Trace Cache would combine instructions from multiple basic blocks into a large basic block).

When the decryption unit is located before the I-cache, it is accessed only on I-cache misses, as shown in Figure 6.4a. Especially for AES decryption, this is essential, since we need to decrypt a whole block (16-bytes) of instructions each time. Decrypting 16-byte blocks at once can lead to reduced power consumption, as the instructions that are executed frequently, e.g.,, in loops, reside decrypted in the I-cache. As shown in Section 6.5.2, even with AES, the runtime overhead is acceptable due to the locality of the I-cache accesses.

To recap, we selected to place the decryption unit after the I-cache for cipher algorithms that can be completed in one-cycle, such as XOR and Transposition. For more complicated cipher algorithms, such as AES, the decryption is placed before the I-cache.

Architectural extensions

AESASISTrequires two extra registers to store the encryption keys: *usrkey* and *oskey*. These registers are memory mapped using a new Address Space Identifier (ASI), and are accessible only by the operating system through two privileged SPARC instructions: *sta* (store word to alternate space) and *lda* (load word from alternate space). The operating system sets the *usrkey* register using *sta* with the key of the user-level process that is scheduled for execution before each context switch. In case of a 32-bit key, a single *sta* instruction can store the entire key. For larger keys, more *sta* instructions are needed.

The AESASIST processor chooses between *usrkey* and *oskey* for decrypting instructions based on the value of the *Supervisor* bit. The Supervisor bit is 0 when the processor executes user-level code, so the *usrkey* is used for decryption, and it is 1 when the processor executes kernel's code (supervisor mode), so the *oskey* is selected. When a trap instruction is executed (*ta* instruction in SPARC), control is transferred from user to kernel and the Supervisor bit changes from 0 to 1; interrupts are treated similarly. Thus, the next instructions will be decrypted with *oskey*. Control is transferred back to user from kernel with the *return from trap* instruction (*rett* in SPARC). Then the Supervisor bit becomes 0 and the *usrkey* is used. The context switch is performed when the operating system runs, and *oskey* is used for decryption. Then the proper key of the process that will run immediately after *rett* is stored at *usrkey*.

Figure 6.3 presents a high level overview for ISR support when using the AES cipher algorithm, and the corresponding MMU that is required. As we can see, every time the I-cache requests an address, the AESASIST controller will request the 16 byte aligned memory block (four LS bits of the address equal to zero) from the AHB interface. This is essential since the I-cache deploys critical word first algorithm, i.e., the address which caused the cache miss will be requested first. The AESASIST controller will request and fetch each word of the cache line using the AHB protocol and fill a 128-bit buffer. When the whole block is fetched the buffer is forwarded along with the decryption key to the AES decryption unit. The AES decryption unit we deployed expands the key on each round and requires 12 cycle to decrypt each block. When the block of instructions is decrypted each instruction is forwarded to the I-cache controller in the same manner the vanilla processor's AHB interface would, hence we preserve the critical word first policy. While these extra steps needed on each I-cache miss seem to impose significant performance overhead, our evaluation results indicate that we can amortize a vast percentage of the overhead in most cases due to the I-cache locality.

Decryption Algorithms and Key Size

Figure 6.5a shows the implementation of XOR decryption with 128-bit key. Since each encrypted instruction in our architecture is a 32-bit word, we need to select the proper 32-bit

(a) Decryption before the I-cache

(b) Decryption after the I-cache

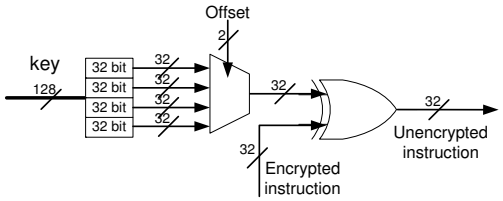
Figure 6.4: Alternative choices for the placement of the decryption unit in the AESASIST-enabled processor.

part of the 128-bit key, the same part that was used in the encryption of this instruction. Thus, we use the two last bits of the instruction's address to select the correct 32-bit part of the 128-bit key using a multiplexer, and finally decrypt the instruction. The same approach is used for XOR decryption with other key sizes, multiple of 32 bits. The implementation of decryption with Transposition, as shown in Figure 6.5b, requires more hardware. This is because it needs 32 multiplexers, one per bit of the decrypted instruction. Each multiplexer has 32 input lines with all the 32 bits of the encrypted instruction, to choose the proper bit.

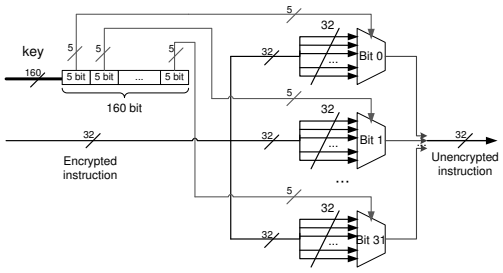
Using AES we have increased area overhead due to the fact that we added an AES decryption unit in the processor. However, modern processors are equipped with cryptography accelerators thus, we can safely consider it a useful addition. For AES key we utilize the 128 least significant bits of the Transposition key. Our AES decryption unit, expands the key per-round. It also has 5 select lines that define the selection of the input bit at each position. The select lines of each multiplexer are part of the 160-bit key. Besides the additional hardware, the runtime operation of Transposition is equally fast with XOR, as it does not spend extra cycles. When using AES the runtime overhead is slightly increased due to the complexity of the encryption algorithm, i.e. 12 cycles per 16 byte block instead of on-cycle for XOR and trasposition. To dynamically select the decryption algorithm and key size, we have added a memory mapped register: *asist_mode*.

Return Address Encryption

Return address encryption was first introduced in the initial ASIST implementation [144], as well as in other proposed mechanisms [69, 202], in order to prevent the disclosure of the text pages location. Encryption in those mechanisms is applied to all code pointers. In our new design, we are not only relying on code pointer obfuscation for preventing gadget discovery; by encrypting the text segment with a strong encryption scheme (AES), even the disclosure of code pointers is not sufficient for an attacker to discover usable gadgets. Finally, in the case where an attack can be mounted without disclosing the text segment of the process (e.g. indirect JIT-ROP), any binary obfuscation technique can be used along



(a) *Decryption using XOR with 128-bit key.* Based on the last two bits of the instruction's address (offset) we select the respective 32-bit part of the 128-bit key for decryption.



(b) *Decryption using Transposition with 160-bit key.* The implementation needs 32 multiplexers with all the 32 bits of the encrypted instruction as input lines in each one.

with our ISR with strong encryption in order to provide sufficient protection. Such attacks require extensive knowledge of the victim application in order to succeed [75].

6.3.3 Operating System Support

Kernel Modifications

In our prototype we modified the Linux kernel, and we ported our changes to 2.6.21 and 3.8 kernel versions. First, we added two new fields in the process table records (`task_struct` in Linux kernel): the process's *key* and the *asist_mode*. We initialize the process's *key* to zero and *asist_mode* to dynamic, so each unencrypted program will be dynamically encrypted.

We changed the binary ELF loader to read the key of the executable ELF file, in case it is statically encrypted, or generate a random key, in case of dynamic encryption, after calling the `execve()` system call. Then, the loader stores the process's key to the respective process table record. We also changed the scheduler to store the key of the next process that is scheduled to run in the *usrkey* register before *each* context switch. For this, we added an *sta* instruction before the context switch to store a 32-bit key. For larger keys, the number of *sta* instructions depends on key size.

To implement dynamic encryption and shared library support we modified the page fault handler. For each page fault, we check whether it is related to code (text page fault) and whether the process that caused the page fault uses dynamic code encryption. If so, we allocate a new anonymous page that is not backed by any file. Upon the reception of the requested page from disk (or buffer cache), we encrypt its data with process's key and copy it at the same step into the newly allocated page. Then, the new page is mapped into the process's address space.

Kernel Encryption

To encrypt kernel's code we used the same approach with static binary encryption. We modified an uncompressed kernel image by (*i*) adding a new note section that contains the kernel's encryption key, and (*ii*) identifying and encrypting all code sections. We had to separate code from data into different sections while building the kernel image. The *oskey* register saves the key of kernel's encrypted code. We also modified the bootloader to read and store the kernel's key into the *oskey* register with a *sta* instruction, just before the kernel's execution. Since *oskey* is initialized with zero, which has no effect in XOR decryption that is also default, the unencrypted code of the bootloader can be successfully executed in the randomized processor. In case of AES encryption, if the key is zero, our decryption unit will not decrypt fetched instructions.

We decided to statically encrypt the kernel's code so as to not add extra delay to the boot process. Due to this, the key is decided at the time the kernel image is built and encrypted, and it cannot change without re-encryption. Another option would be to encrypt the kernel's code while booting, using a new key that is randomly generated per boot. This option could delay to the boot process. Most systems typically use a compressed kernel image that is decompressed while booting. Thus, we can encrypt the kernel's code during the kernel loading stage when the image is decompressed into memory. The routine that decompresses and loads the kernel to memory must first generate a random key, then encrypt the kernel's code along with decompression and store the key in the `oskey` register.

6.4 AESASIST Prototype Implementation

In this section we describe the AESASISTprototype, and present the results of the hardware synthesis using a FPGA, in terms of additional hardware needed compared to the unmodified processor. We also discuss how the proposed system can be ported to other architectures and systems.

6.4.1 Hardware Implementation

We implement AESASISTon Leon3 SPARC V8 processor [4], a 32-bit open-source synthesizable processor [68]. Leon3 uses a single-issue, 7-stage pipeline, 8 register windows and a 16 KB 2-way set associative D-cache. In order to study the overhead which will be imposed in a typical processor we also configured our Leon3 with 16 KB (2-way associative), 32 KB and 64 KB (4-way associative) I-cache sizes respectively. The cache has a single level and is pure Harvard architecture. Finally, we synthesized and mapped the modified AESASIST processors on a Xilinx XUPV5 ML509 FPGA board [204]. The FPGA has 256 MB DDR2 SDRAM memory and operates at 80 MHz clock frequency. The source code of our implementation is available at [71].

Table 6.1: Additional hardware used by AESASIST. We see that AESASIST adds just 0.6%–0.7% more hardware with XOR decryption using a 32-bit key, while it adds significantly more hardware (6.6%–6.9%) when using Transposition. When using AES the overhead is slightly over 10%

Synthesized Processor	Flip Flops	LUTs
Vanilla Leon3	9,227	16,986
XOR with 32-bit key	9,294 (0.73% increase)	17,090 (0.61% increase)
XOR with 128-bit key	9,486 (2.81% increase)	17,116 (0.77% increase)
Transposition with 160-bit key	9,838 (6.62% increase)	18,153 (6.87% increase)
AES with 128-bit key	10,207 (10.6% increase)	18,405 (8.3% increase)

6.4.2 Additional Hardware

Table 6.1 shows the results of the synthesis for three different hardware implementations of AESASIST: (i) using XOR decryption with 32-bit and 128-bit keys respectively, (ii) using decryption with Transposition and a 160-bit key, and (iii) using AES decryption with 128-bit key. We also show the case for the unmodified Leon3 processor, as a baseline to measure the additional hardware used by AESASIST to implement the ISR functionality in each case. We see that AESASIST with XOR encryption and 32-bit key adds less than 1% of additional hardware, both in terms of additional flip flops (0.73%) and lookup tables (0.61%). When a larger key of 128 bits is used for encryption, we observe a slight increase in the number of flip flops (2.81%) due to the larger registers needed to store the two 128-bit keys. The implementation of Transposition results in significantly more hardware used, both for flip flops (6.62% increase) and lookup tables (6.87% increase). This is due to the larger circuit used for the hardware implementation of Transposition, which consists of 32 multiplexers with 32 input lines each, as we showed in Section 6.3.2. Finally, the AES implementation results in approximately 10% more hardware, mostly due to its larger complexity. Obviously, the extra hardware required can be decreased in cases where the processors already contain cryptographic accelerators (such as the AES-NI [98] contained in the majority of recent Intel and AMD CPUs). Such accelerators can be used directly by our ISR implementation, thus amortizing a large percentage of the area overhead.

6.4.3 Kernel and Software Modifications

As we describe in Section 6.3.3, we have modified the Linux kernel and its tool-chain in order to provide a full-featured SPARC workstation. In particular, we ported our Linux kernel modifications in 3.8.0 kernel version. We built a cross compilation tool chain with `gcc` version 4.7.2 and `uClibc` version 0.9.33.2 to cross compile the Linux kernel, libraries, and user-level applications. Thus, all programs running in our system (both vanilla and

AESASIST), including the vulnerable programs that we use for the security analysis, as well as the benchmarks that we use for the performance evaluation, were cross compiled on another PC. We created a new linker script to separate code and data for both static and dynamic code encryption, and align headers, code, and data into separate pages in case of dynamic encryption.

6.4.4 Portability to Other Architectures

Complex instruction set computer (CISC) and 64-bit architectures

In our current prototype, we have implemented the runtime decryption of instructions for RISC architectures that use fixed-length instructions. Thus, porting the decryption functionality in other RISC systems is straightforward. Moreover, our design does not require any modification in the standard registers and data path, hence it can easily tailored to 64-bit (or wider) architectures in which the instruction length is still 32-bit wide, e.g., SPARC, RV64I, etc. In the case of larger instruction width, the only downside would be the number of instructions contained in each 16-byte encrypted block. CISC architectures, such as x86, support variable-length instructions. In our design, we encrypt instructions per 16-byte blocks, thus we do not depend on the instruction length. Moreover, since the instructions are stored decrypted in the I-cache, we do not interfere with the split-line access technique utilized in CISC architectures (i.e., when an instruction is split in two different cache lines).

Regarding AESASIST's hardware extensions, implementing new registers that are accessible by the operating system is straightforward in most architectures, including x86. Encrypting the return address at each function call and decrypting it before returning depends on the calling convention at each architecture. For instance, in x86 it can be implemented by slightly modifying *call* and *ret* instructions. Finally, even though we have implemented our prototype by modifying the Linux kernel, the same modifications (i.e., binary loader, the process scheduler and the page fault handler) can be made in other operating systems as well.

Out-of-order execution

The Leon3 processor that we use for the implementation of AESASIST is a simple 7-stage pipelined processor. As such, there are no options available to synthesize an out-of-order core. Even though this prevents us from evaluating AESASIST in terms of performance and circuit area overhead, still we believe that our design can be used in out-of-order execution cores without any modifications. This is due to the fact that AESASIST can decrypt every instruction right before being stored in the I-cache, hence not modifying the processor core itself. In terms of performance overhead, we can extrapolate that the resulting

overheads will be similar with the in-order execution which are presented in Section 6.5.2. The overhead of AESASIST highly correlates with the I-cache miss rate, rather than the core's architecture. In terms of area overhead, we can expect that our additional hardware will be a lower percentage of the processor than the current, in-order, implementation of AESASIST since out-of-order cores require more circuity area.

6.5 Experimental Evaluation

We now present the experimental evaluation of our AESASISTprototype in terms of security and performance. As described in Section 6.4, our prototype is implemented on a FPGA, running Linux kernel v2.6 or v3.8. For the security evaluation, we also configured the networking of our base system and the corresponding Etherner adapter, in order to enable remote exploitation attempts. We also install a `ssh` server in order to connect to the Linux OS and execute the cross-compiled benchmark programs. The output of each benchark is redirected to local files, thus avoiding any network delays.

6.5.1 Security Evaluation

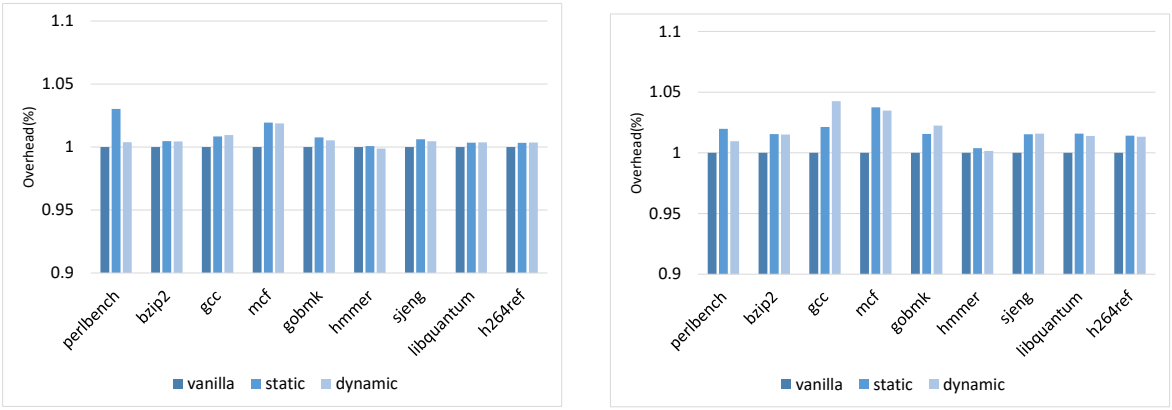
Synthetic Attacks

In our first experiment, we use a vanilla v.2.6.21 kernel, which does not properly implement a non-executable stack on SPARC. We build a custom program with a typical stack-based buffer overflow vulnerability, and we use a large command-line argument to inject SPARC executable code into the program's stack, which successfully executes after overwriting the return address. We then use an AESASIST modified kernel without enabling the return address encryption, and we run a statically encrypted version of the vulnerable program with the same argument. In this case, the program is terminated with an illegal instruction exception, as the unencrypted injected code cannot not be executed. Similarly, we run an unencrypted version of the vulnerable program and relied on the page fault handler for dynamic code encryption. Again, the injected code caused an illegal instruction exception due to the ISR.

Real Attacks

To demonstrate the effectiveness of AESASIST at preventing real code injection attacks that exploit user- or kernel-level vulnerabilities, we test attacks, shown in Table **??**. The first six attacks target buffer overflow vulnerabilities on user-level programs, while the last three attacks, a `NULL` pointer dereference and two buffer overflow vulnerabilities in kernel space.

In addition, we performe similar tests with other known vulnerable programs: Ettercap,



(a) XOR

(b) Transposition

Figure 6.6: Percentage of overhead of (a) XOR and (b) Transposition when using the SPEC CPU2006 benchmark suite. We see that both ASIST implementations have negligible runtime overhead compared to the vanilla system.

which is a packet capture tool, MariaDB database, sendmail, Light HTTPd and Null HTTPd webservers. These programs were cross compiled with our toolchain and encrypted with our extended `objcopy` tool. The injected shellcode is executed successfully only on the vanilla system, while AESASIST always prevents the execution of the injected code and results in an illegal instruction exception.

6.5.2 Performance Evaluation

To evaluate the performance of AESASIST, we run the `SPEC CPU2006` benchmark suite and two real world applications, in three different setups: (*i*) a vanilla Leon3 with unmodified Linux kernel (namely *Vanilla*), (*ii*) AESASIST with static encryption (namely *AESASIST-Static*), and (*iii*) AESASIST with dynamic code encryption (namely *AESASIST-Dynamic*).

Micro-Benchmarks

In our first benchmark, we run a representing subset of the integer benchmarks (CINT2006) from the SPEC CPU2006 suite [180], which includes several CPU-intensive applications. Figure 6.6 and Table **??** shows the slowdown of each benchmark when using AESASIST with static and dynamic encryption respectively, compared to the vanilla system. We observe that both XOR and Transposition impose less than 3% slowdown in all benchmarks. This is due to the hardware-based instruction decryption, which does not add any observable delay. Moreover, the modified kernel performs only minor extra tasks, such as reading the key from the executable file (for static encryption) or randomly generating a new key

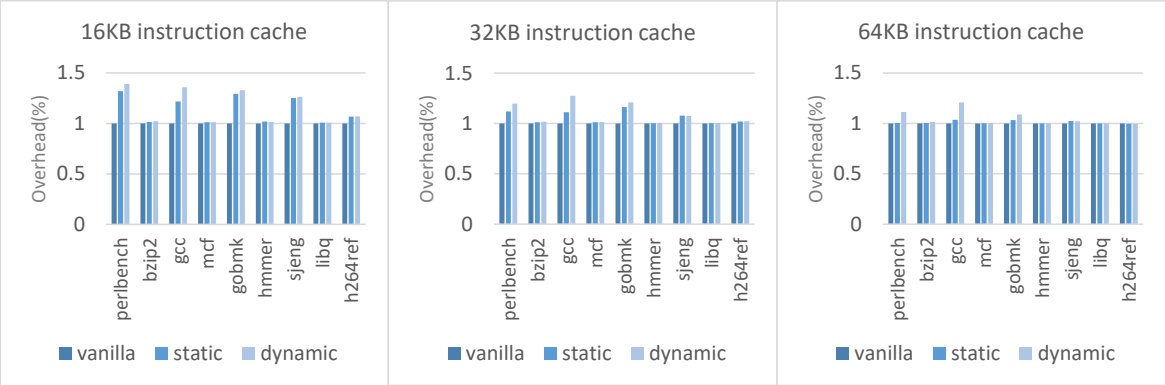


Figure 6.7: Percentage of overhead with AES when utilizing different I-cache sizes. We see that AESASIST with AES imposes significant overhead when the I-cache size is relatively small and thus the ASIST unit is invoked frequently. When we increase the I-cache size, the runtime overhead is reduced to practical margins. Note that the maximum I-cache size we used in our configurations, is typical for modern commodity processors.

(for dynamic encryption) once per each execution, while adding only one extra instruction before each context switch. We notice a slight deviation from the vanilla execution time only for three of the benchmarks: `gcc`, `sjeng`, and `h264ref`. For these benchmarks, we observe a slowdown of 1%–1.2% in static and 1%–1.5% in dynamic encryption, which is probably due to the different linking configurations (statically linked versus dynamically linked shared libraries).

One might expect that the dynamic encryption approach would experience a considerable performance overhead due to the extra memory copy and extra work needed to encrypt code pages at each text page fault. However, our results in Figure 6.6 indicate that dynamic encryption performs equally well with static encryption. Thus, our proposed approach to dynamically encrypt program code at the page fault handler, does not seem to add any extra overhead.

For the AES implementation, we observe in Figure 6.7 and Table **??** an increased overhead; this is due to the fact that $\tilde{1}2$ cycles are added on each I-cache miss. This overhead is further exaggerated when using dynamic encryption. The performance impact when using the default I-cache size (i.e. 16KB) is quite impractical for some of the benchmarks. When using larger I-cache though, the performance impact of both static and dynamic solutions drops significantly—for a 64 KB I-cache, only `gcc` and `perlbench` exceeded 10% runtime overhead, due to their high I-cache miss rate. Thus, we can safely assume that even when using AES encryption, ISR is a practical solution.

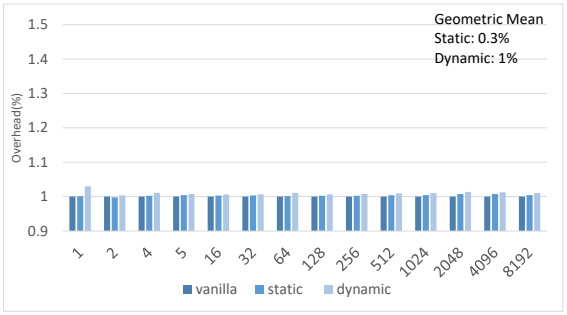
Table 6.4: Data and text page faults per second and percentage during execution when running the SPEC CPU2006 benchmark suite. Text page faults rarely occur, attributing to less than 5% of the total page faults in most benchmarks. This explains the negligible overhead of the dynamic encryption approach.

Benchmark	Data page faults rate	Text page faults rate	Data page faults (%)	Text page faults (%)
400.perlbench	38.4964	1.97215	95.1267%	4.87329%
401.bzip2	44.3605	0.193831	99.565%	0.435044%
403.gcc	60.3235	3.93358	93.8784%	6.12164%
429.mcf	51.7769	0.0497679	99.904%	0.0960275%
445.gobmk	25.4735	0.905984	96.5656%	3.43442%
456.hmmer	0.0546246	0.0223249	70.9877%	29.0123%
458.sjeng	71.9751	0.0676988	99.906%	0.0939702%
462.libquantum	5.18675	0.0486765	99.0702%	0.929752%
464.h264ref	3.19614	0.0333707	98.9667%	1.03331%

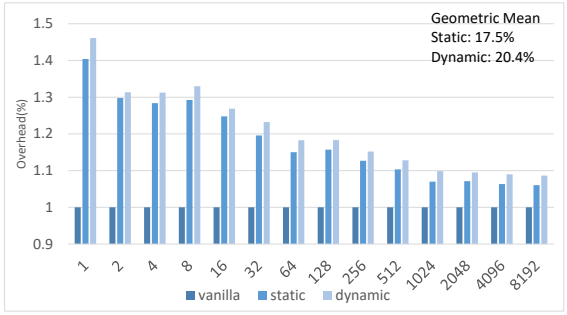
To better understand the performance of this approach, we further instrumented the Linux kernel to measure the data and text page faults of each process that uses the dynamic encryption mode. Table 6.4 shows the data and text page faults per second for each benchmark. We see that all benchmarks have a very low rate of text page faults, and most of them experience significantly less than one text page fault per second. Moreover, we observe that the vast majority of page faults are for data pages, while only a small percentage of the total page faults are related to code. The negligible overhead with dynamic code encryption at the page fault handler is due to two main reasons: (*i*) as we see in Table 6.4, text page faults are very rare, and (*ii*) the overhead of the extra memory copy and page encryption is significantly less that the page fault's overhead for fetching the requested page from disk. Note that in our setup we use a RAM file system instead of an actual disk, so a production system may experience an even lower overhead. The very low page fault rate for pages that contain executable code makes the dynamic encryption a very appealing approach, as it imposes practically zero runtime overhead, and at the same time it supports shared libraries and transparently generates a new key at each program execution.

Real-world Applications

First, we run the `lighttpd` web server in a vanilla system and in the two encryption approaches (i.e., static and dynamic of AESASIST. Figure 6.8 shows the slowdown of the average download time for different file sizes. We see that AESASIST does not impose any considerable delay, as the download time remains within 1% of the vanilla system for all file sizes. We notice that both static and dynamic encryption implementations perform equally good. We measure the page faults caused by `lighttpd`: 261 data page faults per second, while only 0.013 text page fault per second. Moreover, most of these text page faults occur during the first few milliseconds of the `lighttpd` execution, when the code is



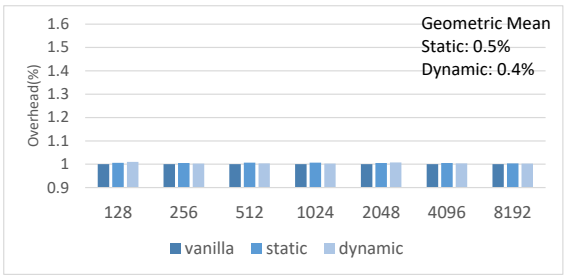
(a) Percentage of overhead when downloading different files from a lighttpd Web server as a function of the file size. We see that AESASIST adds less than 1% delay for all file sizes.



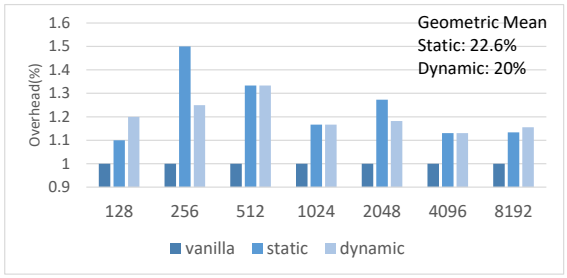
(b) Percentage of overhead when using AES ASIST protected lighttpd downloading different files from a lighttpd Web server as a function of the file size. We can see that for the typical size of HTTP responses the overhead percentage is 9%

Figure 6.8: Percentage of overhead of lighttpd in all ISR modes.

loaded into memory. When running `lighttpd` with AES encryption and a 64 KB I-cache (Figure 6.8(b)), we notice that despite the overhead measured when using small datasets, the overhead is reduced when the HTTP response size exceeds 2 KB. [1]



(a) Percentage of overhead when inserting data into sqlite3 as a function of the number of insertions. We see that AESASIST experiences less than 1% slowdown even for very small datasets.



(b) Percentage of overhead when inserting data into AES ASIST protected sqlite3 as a function of the number of insertions.We see that AES AESASIST experiences approximately 15% overhead when using dynamic encryption.

Figure 6.9: Percentage of overhead of sqlite3 in all ISR modes.

Next, we run a `sqlite3` database using the vanilla and the three AESASIST setups. To evaluate `sqlite3` we implement a benchmark that reads a large text file and updates a SQL table, using the C/C++ SQLite interface. Figure 6.9(a) shows the slowdown when inserting data into the database as a function of the number of insertions. AESASIST imposes less than 1% slowdown on the database's operation for both static and dynamic approaches,

[1] The typical workload size for HTTP responses is well over 2 KB on average, except from MicroBlogs where the average size is 1KB [116].

even on small datasets that do not provide AESASIST with enough time to amortize the encryption overhead. The same behaviour is noticed when we run `sqlite3` with AES encryption and a 64 KB I-cache (Figure 6.9(b)).

6.6 Related Work

Instruction Set Randomization. ISR was initially introduced as a generic defense against code injections by Kc et al. [104] and Barrantes et al. [24, 25]. To demonstrate ISR, they proposed implementations with `bochs` [114] and `Valgrind` [134] respectively. Hu et al. [95] implemented ISR with Strata SDT tool [167] using AES as a stronger encryption for instruction randomization. Boyd et al. [37] propose a selective ISR to reduce the runtime overhead. Portokalidis and Keromytis [151] implemented ISR using Pin [27] with moderate overhead and shared libraries support. In Section 6.6.1 we described in more detail all the existing software-based ISR implementations and we compared them with AESASIST. AESASIST addresses most of the limitations of the existing ISR approaches owing to its simple and efficient hardware support. Polyglot [171] is similar with ASIST, but does not offer dynamic encryption, hence blind ROP attacks are still possible. Shuffler [202] protects only user-space processes by periodically re-randomizing their address space layout. The re-randomization is perfomed by a separate thread that runs in parallel, one for each process. Morpheus [69] also uses code pointer obfuscation and instruction set randomization.

ISR protects a system against any native code injection attacks. To accomplish this, ISR uses per-process randomized instruction sets. This way, the attacker cannot inject any meaningful code into the memory of the vulnerable program. The injected code will not perform the intended malicious behavior and will probably crash after just a few instructions [24]. To apply the ISR idea, existing implementations first encrypt the binary code of each program with the program's secret key before it is loaded for execution. The program's key defines the mapping of the encrypted instructions to the real instructions supported by the CPU. Then, at runtime, the randomized processor decrypts every instruction with the proper program's key before execution. Injected instruction that have not been correctly encrypted will result in irrelevant or invalid instructions after the obligatory decryption. On the other hand, correctly encrypted code will be decrypted and executed normally.

6.6.1 Limitations of Existing Implementations

Existing ISR Implementations use binary transformation tools, such as `objcopy`, to encrypt the code of user-level programs. For runtime decryption they use emulators [114] or dynamic binary instrumentation [27, 134, 167]. In Table **??** we list and compare existing ISR

implementations.

Kc et al. [104] implemented ISR by modifying the `Bochs` emulator using XOR with a 32-bit key in their prototype. The use of an emulator results in significant slowdown, up to 290 times slower execution on CPU intensive applications. Barrantes et al. [24, 25] use `Valgrind` [134] to decrypt applications' code, which is encrypted with XOR and a random key equal to the program's length. This prototype supports shared libraries by copying each randomized library per process, and offers an API for self-modifying code. However, the performance overhead with Valgrind is also very high, up to 2.9 times slower than native execution. Hu et al. [95] implemented ISR with a software dynamic translation tool [167] using AES encryption with 128-bit key size. Dynamic translation results in lower but still significant performance overhead, that is close to 17% on average and as high as 250%. To reduce runtime overhead, Boyd et al. [37] proposed a selective ISR that limits the emulated and randomized execution only to code sections that are more likely to contain a vulnerability. Portokalidis and Keromytis [151] implemented ISR with shared libraries support using Pin [27]. The runtime overhead ranges from 10% to 75% for popular applications, while it has four-times slower execution when memory protection is applied to Pin's code.

Polyglot [171] is similar with ASIST, but does not offer dynamic encryption. Thus, attackers can still launch blind ROP attacks, since the code will not be encrypted with different keys on each launch. Moreover, it uses a separate shared key cache inside the processor, in order to support dynamic libraries, which increases the circuit area overhead of the proposed mechanism significantly. Shuffler [202] proposes a mechanism that periodically re-randomizes the address space layout of a process. The proposed mechanism offers adequate protection against control-flow attacks by continuously modifying the code pointer addresses. However, it is not scalable because it requires a parallel thread to each protected process, responsible for the re-randomization. Moreover, the proposed mechanism can protect only user-level applications. Morpheus [69] is a promising architecture for thwarting the majority of control-flow attacks. Yet, it has been only implemented and evaluated on gem5 simulator [30], with a single process context, using the system call emulation capabilities of gem5, without any estimation of the area overhead of their design. The proposed mechanism requires extensive modifications of the original processor by adding several hardware components, which may be impractical for low-powered or resource-contraint devices (e.g. IoT devices, etc.), due to the extra substantial area overhead Finally, the authors state that the latency of their mechanism will be prominent in pipelined processors; this does not seem to be the case though, since every execution unit of an out-of-order processor should be extended with an attack detector, imposing even more area overhead in the final design. ARM recently launched processors with pointer

[2] Implemented in simulator, with system call emulation.

authentication capabilities [115]. In this approach authentication codes are created for every control-flow pointer, calculated with special instructions. If an adversary modifies pointers residing in the stack, the modification will be detected by the authenticating instructions placed just before the control-flow transition instruction i.e. indirect jump, return. However, Google project zero reviewed the deployment of the PAC and found that forging pointer authentication codes was possible [83].

Defenses against code injection attacks. Modern hardware platforms support non-executable data protection, such as the No eXecute (NX) bit [148]. NX bit prevents data from being executed, so it can protect against code inject attacks without performance degradation. However, its effectiveness depends on its proper use by software. For instance, an application may not set the NX bit on all data segments due to backwards compatibility constraints, self-modifying code and bad programming practices. We believe that AESASIST can be used complementary to NX bit, in case that NX bit may not be applicable or can be bypassed. For instance, many ROP exploits use the code of `mprotect()` to make executable pages with injected code, bypassing the NX bit protection. This way, they can execute arbitrary code to implement the attack without the need of more specific gadgets, which may not be easy to find, e.g.,, due to the use of ASLR. In contrast, these exploits cannot execute injected code in a system using AESASIST, as this code will not be correctly encrypted. Thus, AESASIST with ASLR provides a stronger defense.

Attacks demonstrated by Snow et al. [174] is also able to bypass NX bit and ASLR using ROP. First, it exploits a memory disclosure to map process's memory layout, and then it uses a disassembler to dynamically discover gadgets that can be used for the ROP attack. AESASIST with ASLR, however, is able to prevent this attack: even if memory with executable code leaks to the attacker, the instructions will be encrypted with a randomly-generated key. This way, attacker will not be able to disassemble the code and find useful gadgets. AESASIST ensures that key does not reside in process's user space memory, while the stronger variant which utilizes AES encryption algorithm, aims to avoid inferring the key, even when the ciphertext and plaintext are known. SecVisor [168] protects the kernel from code injection attacks using a hypervisor to prevent unauthorized code execution. While SecVisor focuses on kernel's code integrity, AESASIST prevents the execution of unauthorized code in both user and kernel level.

Defenses against buffer overflow attacks. StackGuard [57] uses canaries to protect the stack, while PointGuard [56] encrypts all pointers while they reside in memory and decrypts them before they are loaded into a register. Both techniques are implemented with compiler extensions, so they require program recompilation. In contrast, BinArmor [173] protects existing binaries from buffer overflows, by discovering the data structures and then rewriting the binary. CFI is a mechanism that aims to limit the locations where a code pointer can point [10]. Many different policies have been proposed to reduce the performance overheads and reduce the analysis required [61, 205]. However, it has been

shown that relaxed CFI policies are not sufficient against code-reuse attacks [53].

Other randomization-based defenses. ASLR [149] randomizes the memory layout of a process at runtime or at compile time to protect against code-reuse attacks. Giuffrida et al. [72] propose an approach with address space randomization to protect the operating system kernel. Bhatkar et al. [29] present randomization techniques for the addresses of the stack, heap, dynamic libraries, routines and static data in an executable. Wartell et al. [199] randomize the instruction addresses at each execution to address code-reuse attacks. Jiang et al. [102] prevent code injections by randomizing the system call numbers.

6.7 Summary

To summarize this chapter, we designed, implemented and evaluated a a hardware-assisted architecture for ISR support, namely AESASIST, which is able to protect both user- and kernel-level processes transparently, without any program modifications. AESASIST uses a combination of techniques to increase security and performance. In particular, it utilizes highly secure cryptographic algorithms, i.e., AES, that can provide resilience against different types of attacks (e.g., known cipher-text and plain-text, anagrams, etc.). Moreover, it takes advantage efficient caching strategies and spatial locality of code in order to decrease the execution overheads. By doing so, it is able to decrease the excessive number of decrypt operations—especially for block ciphers that operate on many bytes at once—and improve the overall performance.

Our experimental evaluation shows that AESASIST imposes marginal overheads (less than 1.5% for XOR and Transposition, and about 10% for AES), while it is able to prevent attacks that exploit both user- and kernel-level memory vulnerabilities. Overall, our work shows that AESASIST can address most of the limitations of existing software-based ISR implementations, and can be easily ported to other hardware architectures to defend against code injection attacks.

,,,,,,

,,,,,,,,,,,

Chapter 7
Control-Flow Integrity

In this chapter we present the design and implementation of fine grained Control-Flow Integrity. We achieved this by extending the instruction set of Leon3 Sparc v8 processor (same as 6). Particularly, in this work, we extended our previous hardware-assisted CFI (HCFI) [49] in order to enhance its granularity and flexibility.

Control-Flow Integrity has been in the recent years a mechanism that all major ISAs (x86, ARM, *etc.*) include as an extension. The experince and lessons learned through this and our previous work are being applied in the RISC-V ISA [21], in order to shape the official specification for CFI in RISC-V architecture. Trnasforming an academic work to an industrial solution is not an easy task, since the design choices are much more restrained in order to preserve functionality and compatibility.

The exploitation threats are constantly evolving. For instance, code-reuse attacks, such as Return-Oriented Programming (ROP) [159] and Jump-Oriented Programming (JOP) [34] can potentially take advantage of memory vulnerabilities and transform them to functional exploits. These techniques do not require any code injections; instead, they re-use existing parts of the program to build the necessary functionality without violating DEP. According to reports, more than 80% of the vulnerabilities are exploited using code-reuse attacks [156].

Code randomization techniques [145] are shuffling the location of the code, in order to make code reuse attacks harder to achieve. Still, even a small information leak can reveal all of the process code and bypass any randomization scheme [175]. In the previous chapter 6 we presented a randomization technique that does not randomize addresses, rather the code of the application, in an attempt to prevent the disclosure of gadget locations. However, this strategy is not always effective, it has been proven that even without the knowing the location of gadgets, attackers can still exploit vulnerable applications [31]. Instead of hiding the code, another way for stopping exploits is to prevent the execution of any new functionality, by employing Control-Flow Integrity (CFI) techniques [9]. An attacker cannot inject code or introduce any new functionality that is not part of the *legitimate* control-flow graph (CFG). Unfortunately, the majority of existing CFI proposals

have still many open issues (related to *accuracy* and *performance*), that hinder its applicability [22].

In this work, we enhanced the granularity and flexibility of our previous design [49]. In our previous design we designed and implemented new hardware instructions dedicated for CFI, and the deployed shadow memory within the processor core. In this work we increased the granularity of CFI (especially in forward-edge situations); moreover we cover a couple of intrinsic situations (including the instrumentation of fall-through functions and indirect jumps, such as `switch` statements, within functions). Performance-wise, the implementation in hardware is the optimal choice; our approach adds less than 1% average runtime and 2% power overhead, making it suitable for embedded systems.

Overall, HCFI is a hardware design that offers a CFI solution that is (*i*) *complete*, since it protects both forward and backward edges, (*ii*) *fast*, 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 processor core. Furthermore, we argue that HCFI is the most complete hardware implementation of CFI so far, supporting many problematic cases (such as `setjmp/longjmp`, recursion, fall-through functions and indirect jumps within functions).

Furthermore, the design of HCFI forms the base for CFI in RISC-V architecture, i.e. the design of the Shadow Stack and Landing Pads (SSLP) extension specification. Compared to HCFI, SSLP is a *complete* design for Control-Flow Integrity which offers in-main-memory shadow stacks for supporting multiple threads, supports user-level threads and offers protection across every privilege level of RISC-V (i.e., machine, supervisor and user).

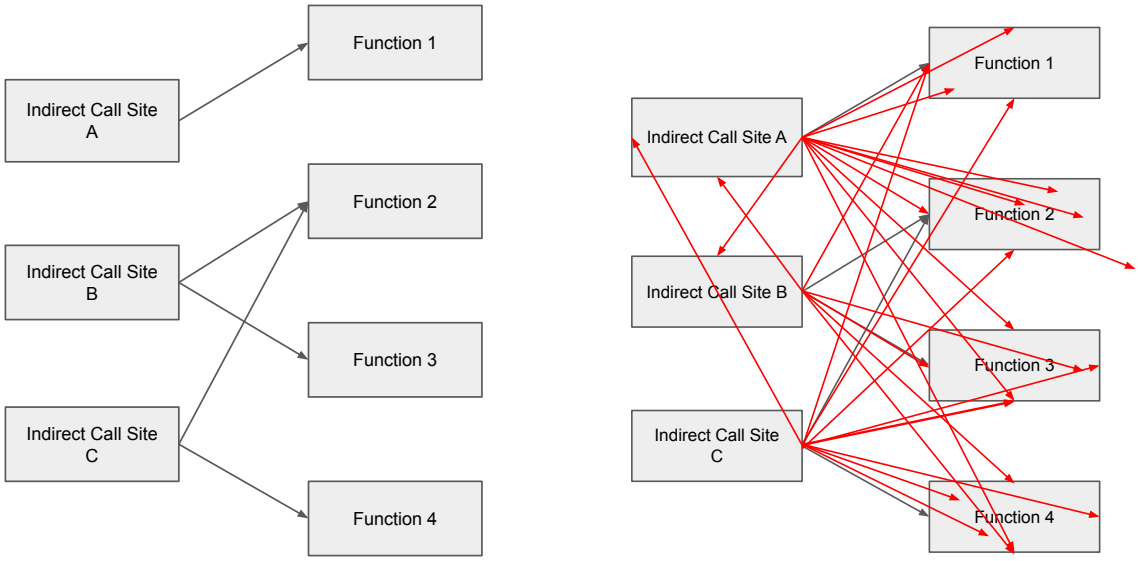
7.1 Background

Control-Flow Integrity (CFI) [9] constraints all indirect branches in a control-flow graph (CFG), which is determined statically before the program execution Fig. 7.1a. In essence, this is achieved by setting a simple set of rules that a program execution flow must adhere to:

1. A call-site “A” can call a function “B” only if the edge (the call itself) is part of the Control-Flow Graph (CFG). This is called Forward-Edge CFI and can easily applied to direct calls, as the only way to modify a direct call is to overwrite the code itself. This is not the case for indirect calls though, where function pointers are typically stored in data regions.

2. A function “B” can only return to the call-site “A” that actually called it, and no other place in the code. This is called Backward-Edge CFI. Backward-edges are, in essence, indirect calls, since they rely on a pointer (return address) to jump to their target.

An attacker cannot inject code (CFI requires that Data Execution Prevention is enabled)



(a) Example of an application CFG

(b) Without CFI indirect jumps can any address.

or introduce any new functionality that is not part of the *legitimate* control-flow graph (CFG). The majority of existing CFI proposals have still many open issues (related to *accuracy* and *performance overhead*), that hinder its applicability [22, 205]. For instance, it is not always easy to compute the program's CFG. This is mainly because the source code might not always be available, while even if it does, dynamic code that might be introduced at run-time or the heavy use of function pointers can lead to inconclusive target resolution [22]. This problem has led researchers to develop CFI techniques that are based on a relaxed approximation of the CFG [205], 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 [73]. 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* [60] - 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 [67]. A trivial case is when a function is called by multiple places in the program. According to the CFG, all return locations are legitimate, however only one is actually correct. Moreover, implementing fine grained CFI solely on software, introduces prohibitive performance impact. In the original CFI proposal by Abadi [9], the average performance was 21%. More recent approaches like SafeStack [111], are designed to offer fine grained backward edge protection with minimal overhead. The applications are instrumented during compilation in order to use a different, protected stack for storing

control flow variables used in backward edges. However, protecting memory regions using software techniques has been proven ineffective against sophisticated attacks [41, 74].

To overcome there restrictions, hardware-assisted CFI implementations can provide architecturally protected memory regions for storing control-flow variables, while at the same time accelerate significantly any checks required during control-flow transitions; this enables the use of fine-grained CFI even in low-powered devices.

7.2 Threat Model

7.2.1 In-scope threats

Our software and firmware threat model assumes that an attacker can exploit a vulnerability, either a stack or heap overflow, or use-after-free, present in the target software binary. This vulnerability can be used to overwrite key components of the running program like return addresses, function pointers, or VTable pointers. We also consider that the attacker has successfully bypassed ASLR, and has full knowledge of the memory layout, CFI is orthogonal to ASLR and does not interfere with it in any way.

Nevertheless, the system enforces that (i) the .text segment is non-writable, preventing the application's code from being overwritten, and (ii) the data segments are non-executable blocking the attacker from executing injected data with proper CFI annotation. With regards to linking we assume that the binaries are configured with RELRO in order to protect GOT and PLT sections from being written. For privileged modes we exclude extending the kernel/firmware using unverified/untrusted kernel/firmware extensions such as drivers or modules that may disable the protections. We also consider that the tool chain used for compiling the binary and libraries loaded by the binary are trusted and will not emit malware behavior intentionally. For each privilege level, we assume that the higher privilege levels are not exploited and that the code is trusted and does not include malware.

7.2.2 Out-of-scope threats

We do not consider Data Oriented Programming (DOP) attacks [94] in our threat model. In this attack scenario the attacker does not need to divert from the predefined control flow graph of the application. Rather, the attacker overwrites (non-control) data in order to change the CFI-compliant behavior of the application, e.g. change the arguments passed to a system call. This exploitation technique is possible even in the theoretically most accurate control flow integrity scheme. In some cases the attacker needs to also combine DOP and code reuse attacks in order to achieve exploitation. Thus, Control Flow Integrity can harden the application against exploitation, since an attacker has a limited set of valid target functions and cannot dynamically select to execute unintended code blocks of the

Table 7.1: Instructions needed to support HCFI.

`SetPC`	Pushes the current program counter (PC) in the shadow stack
`CheckPC`	Pops the shadow stack and compares the result with the next PC
`SetPCLabel`	Can push the PC onto the shadow stack and carries a label used to verify forward edges which is stored in a dedicated register (Label Register). Finally, it sets the requirement the next instruction must be a CheckLabel
`CheckLabel`	Carries a label that is compared to the one in the Label Register
`SJCFI`	Sets the environment for a future longjmp and acts as a landing point for an executing one
`LJCFI`	Signifies that a longjmp is underway

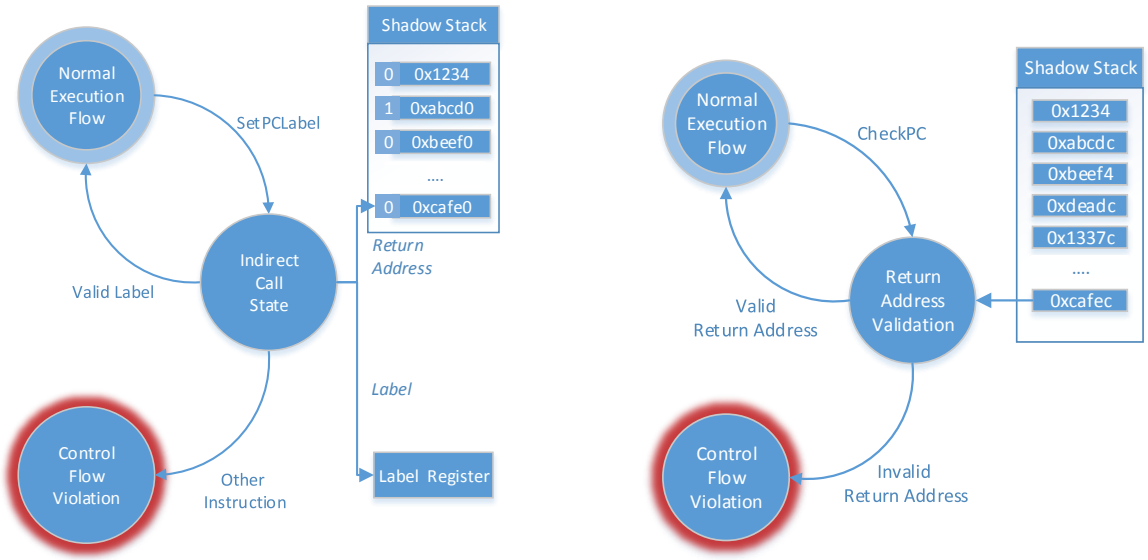
application. We also exclude techniques based on debugging, emulation and code injection using hooking techniques. Finally, side-channel techniques allowing arbitrary data accesses are outside of the problem description CFI aims to solve.

7.3 Hardware-Enforced Control-Flow Integrity

HCFI enforces the set of CFI rules (described in Section 7.1) in hardware, while also provide workarounds for certain corner cases. More specifically, a valid call requires that the call site and the destination have been previously acknowledged to be a valid pair in the CFG. A simple way to avoid checking a list of valid pairs for every indirect call, is to group valid pairs with a label. If the label of the source and the destination match, then the edge is legal.

On the contrary, a valid return is typically simpler to validate. Whenever a call takes place, the return address is pushed to the stack. If the address reached after a return, matches the *top* of the stack, the return is valid. This is achieved by also pushing the return address to a new, hidden, stack (namely *shadow stack*), and comparing the return's target to the one stored at the top of the shadow stack. However, this is not the case for the `setjmp/longjmp` case, in which a function does not necessarily return to its caller. In particular, `longjmp` never returns to its caller but to its matching `setjmp`.

To support this functionality, the ISA is extended with new instructions (shown in Table 7.1): two for the instrumentation of the backward edges, two for the forward edges, and two for handling `setjmp/longjmp` cases. The instructions are strategically placed, so as to wrap the Control-Flow edges. `SetPC` and `SetPCLabel` are paired with direct and in-



(a) FSM for indirect call instructions

(b) FSM for return instructions

Figure 7.2: The basic FSMs for the hardware-based CFI. For return instructions, the target Program Counter is compared with the top value of the stack every-time a `CheckPC` instruction is received and the execution continues nor-mally.

direct calls respectively, while `CheckPC` is paired with return instructions, and `CheckLabel` is placed in function entry points, if the function is an indirect call target. Finally, `SJCFI` and `LJCFI` are paired with the calls to `setjmp` and `longjmp` themselves. `LJCFI` is placed immediately before the call to longjmp, while `SJCFI` is placed immediately after the call to `setjmp`, so that it will be the first instruction executed after a return from `setjmp`, no matter if `setjmp` or `longjmp` was called.

Finally, given that the design of HCFI does not track stack frames, but specific addresses instead, recursion may result in the same address being pushed to the shadow stack multiple times. From this observation, a very simple optimization can be implemented; namely, not storing the address when it equals the top of the stack, but instead marking the address at the top as recursive. This effectively negates the spacial requirements of immediate recursion. During `CheckPC` execution, if the top address in the shadow stack is marked as recursive and is the same as the target of the return instruction it will not be popped. If not, the top address will be popped and the target address will be compared with the next top address in the shadow stack. If the two addresses are equal, the execution will continue normally and the top of the shadow stack will be popped (if the address was not marked as recursive). If the addresses are not equal, CheckPC will result

in a CFI violation.

7.4 Fine-grained CFI Instrumentation

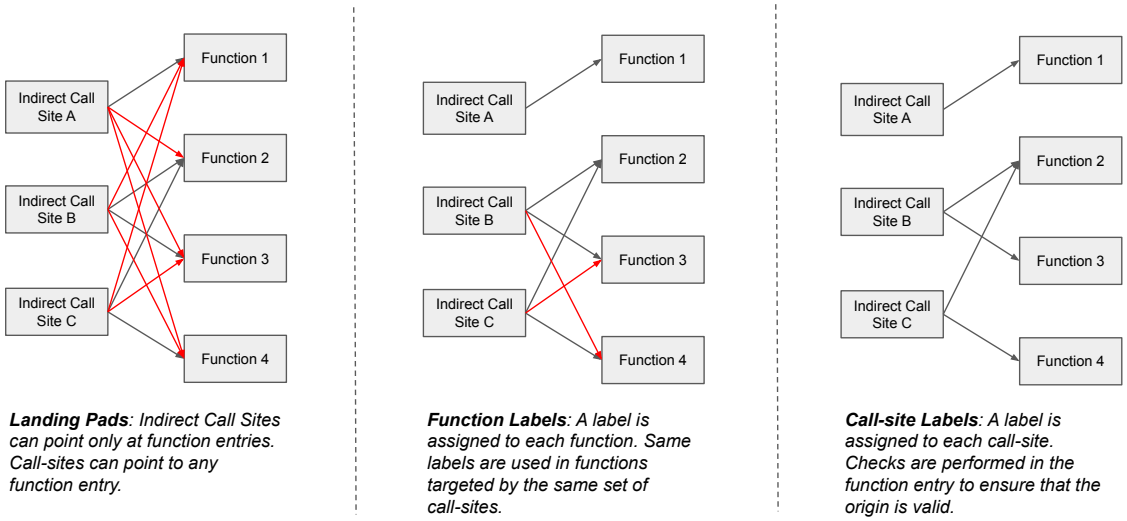


Figure 7.3: Examples of CFG representation based on the granularity offered by the CFI strategy, our previous design offered the per-function CFG (center). Our revised design offers per-indirect-call labels (right-most).

The instructions presented in Section 7.3 are created in order to enable a policy agnostic CFI mechanism. Especially for the backward edges, they can easily support the finest possible granularity: by using an architecturally protected shadow stack where only the CFI instructions can modify values, we can ensure that a function will always return to the original call site. However, for forward edges, the granularity is proportional to the effort of analysis performed on the code of the executable. Ideally, every function in the binary will be reachable by a minimum set of indirect call sites. We note that our design can even support more relaxed forward-edge schemes, where indirect call sites can target every function entry point, i.e. by using only one label in the whole binary — this can be practical in cases, where extensive control flow analysis is not feasible.

To allow for finer granularity and flexibility Fig. 7.3, we make the following modifications to our initial design. Previously, every `CheckLabel` instruction was requiring the Label Register to be set, and hold the correct label. Under the new design, an unset Label Register, or one carrying an incorrect label, does not lead to a violation, as long as the next instruction is also a `CheckLabel`. Also, the `SetPCLabel` instruction can now omit pushing the PC to the shadow stack, depending on its arguments. Moreover, we allow the

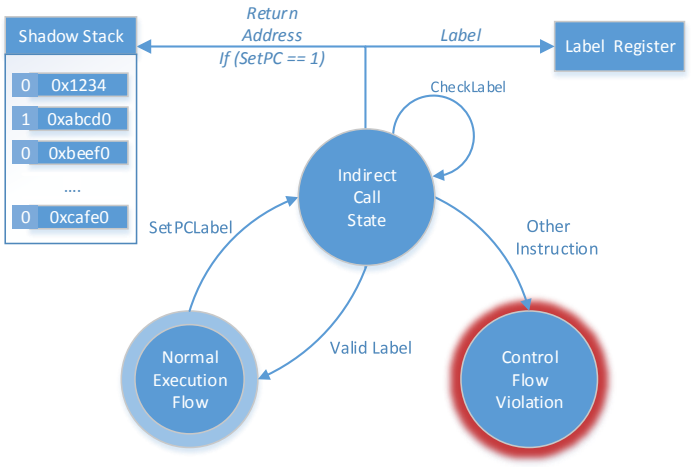


Figure 7.4: The extended FSM for Indirect Call States. A `SetPCLabel` instruction is received, the appropriate memory modules are set, and the core enters a state where only `CheckLabel` instructions are accepted. Once a `CheckLabel` instruction with the appropriate label is received, the execution returns to its normal flow.

instrumentation of indirect branching within the same function. Ignoring `CheckLabel` instructions does not raise security concerns, if the whole binary is instrumented properly. Forward-edge transitions should only be checked during indirect call and branch instructions — during normal execution, the `CheckLabel` instructions do not need to make any checks, since the control-flow is not influenced by data.

7.4.1 Finer Forward-Edge Granularity

When Control Flow Integrity was first introduced by Abadi et. al. [9], indirect call targets with a common source had to be grouped together. For example, if a call site "A" indirectly called a call target "B", and a call site "C" could indirectly call "D" *and* "B", then both call sites "A" and "C", as well as the call targets "B" and "D", would have to share the same label. This is a usual case in C++ applications where indirect calls, dereference virtual table pointers. Target functions that are common between indirect call sites, will force the use of the same label across a large portion of the application. Thus, the granularity of forward-edge protections become significantly coarser.

In this work, we offer the option to set a unique label for each indirect call site, and add as many `CheckLabels` in the call target as needed. The previous example can now be instrumented with 2 labels in the "B" entry point (one for each indirect call-site). Call site "A" and "C" will carry different labels in their CheckLabel instructions. This has the effect

of not allowing call site "A" to jump to "D", which was previously possible. This allows for much finer forward-edge CFI on top of an already powerful design. Figure 7.4 presents the operation of `CheckLabel` instruction.

Fall-through Functions

In many popular libraries, such as GNU libc, there are functions with overlapping code sections [13]. In such cases, the execution of a function falls-through into another function's entry point (without using branch instructions). If these functions are possible targets of indirect call instructions, they should be instrumented with `CheckLabel` instructions, otherwise even if the indirect transition is valid it will result to a CFI violation. Since `CheckLabel`s do not cause a CFI violation when the processor is not in indirect jump state, they are just ignored during execution. Thus, when a function falls through, the execution of the inner function's `CheckLabel` instructions will not result in a CFI violation. This allows for fall-through functions to be instrumented like regular functions.

Intra-Function Forward-Edges

Most CFI schemes do not take into account indirect branches, targeting addresses within the same function. For example, large `switch` statements are usually compiled to jump tables in order to reduce the code size of the binary. In these cases the address of each case is stored in a jump table. At runtime, the result of the `switch` statement is used in an indirect jump in order to dereference the jump table at the appropriate index. Thus, instead of emitting absolute jumps for every possible statement result, the compiler emits a single indirect jump that uses the statement result as an index in the jump table. In our design we offer the capability to instrument those indirect jumps in order to ensure that the target address is the entry point of one of the cases. Each indirect jump will be instrumented with a `SetPCLabel` instruction that will not push a return address in the shadow stack (i.e. `SetPC` bit is '0'), and the entry points of each *case* basic block will be instrumented with the appropriate `CheckLabel` instruction. Every `switch` statement in the binary should use a different label for better granularity.

7.5 Implementation

To implement the hardware-based CFI described in the previous sections, we extended the Leon3 SPARC V8 processor, which is a 32-bit open-source synthesizable processor. Overall, the additions to the core can be grouped in the following two categories: (*i*) Memory Components and (*ii*) CFI Pipeline.

7.5.1 Memory Components

The following new memory components are deployed in the Register File of the core:

- A 256*32 bit dual-port Block RAM was used for the Shadow Stack.
- A 256*8 bit single-port Block RAM was used for the `setjmp` and `longjmp` support (SJLJRAM).
- A 18 bit register was used to store the label for forward edge validation (Label Register).
- A 256*1 bit array helped us optimize recursive calls (Recursion Array).

7.5.2 CFI Pipeline

Our instructions enter the Integer Unit's (IU) pipeline as usual, however they do not interfere with it. We have developed a new pipeline within the IU (CFI Pipeline) that operates in parallel and provides the functionality required everytime the instructions are decoded.

- `SetPC` first tops the Shadow Stack and compares it to the current Program Counter (PC). If the memory addresses match, the Recursion Array is set; otherwise, the address is pushed onto the shadow stack. In case the Shadow Stack is full a *Full* violation is raised.
- `SetPCLabel` is in essence two instructions, meaning that it acts exactly as a `SetPC` and what could be described as a `SetLabel`. The `SetPC` functionality works only if the 25th LS bit of the instruction is set. Regardless of the `SetPC` functionality, the Label carried in its 18 LS bits is written to the Label Register, and the CFI Pipeline transitions to the `SetLabel` state. This mandates that only `CheckLabel` instructions can be executed, until one with the correct label is issued. If any other instruction is issued, a *Control Flow* violation is raised.
- `CheckLabel` compares the Label carried in its 18 LS bits to the label stored in the Label Register, if the CFI Pipeline is in the `SetLabel` state. Otherwise, it is ignored and acts as a `nop`. If the comparison holds, the Label Register is reset and the pipeline transists from SetLabel state to normal execution. If not, the execution continues, but if an instruction other than checklabel is issued, a *Control Flow* violation will be raised.
- `CheckPC` first checks the Shadow Stack; if it is empty, an *Empty* violation is raised. Otherwise, it tops the Shadow Stack, increments the value by four (one instruction) and compares it to the next PC. If the addresses match and the equivalent recursion bit is not set, the Shadow Stack is popped. If the addresses did not match but the

recursion bit is set, the address is popped and another comparison is performed with the next value. Again, if they match and the top value is not recursive, it is popped. If the first comparison failed and the top address was not recursive, or if both comparisons failed, a *PC Mismatch* violation is raised.

- `SJCFI` changes its functionality depending on whether the CFI Pipeline is in the `longjmp` state. If it is not, it writes the current depth of the Shadow Stack to the SJLJRAM. The address is provided by a label it carries on its 8 LS bits. Otherwise, it uses the same label to read the address from the SJLJRAM and set the Shadow Stack to that depth. The Shadow Stack will not allow an index higher than the current, so that previously popped addresses cannot be abused. The CFI Pipeline returns to its default state.

- `LJCFI` sets the pipeline in the `longjmp` state until an `SJCFI` instruction is executed.

7.6 Performance Evaluation

We synthesize and program our new design, based on the Leon3 soft-core, 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. Since we are targeting embedded systems, we run all tests without an operating system present. We instrumented most of the SpecInt2000 suite and a few microprocessor benchmarks, namely Coremark, Dhrystone, and Matmul.

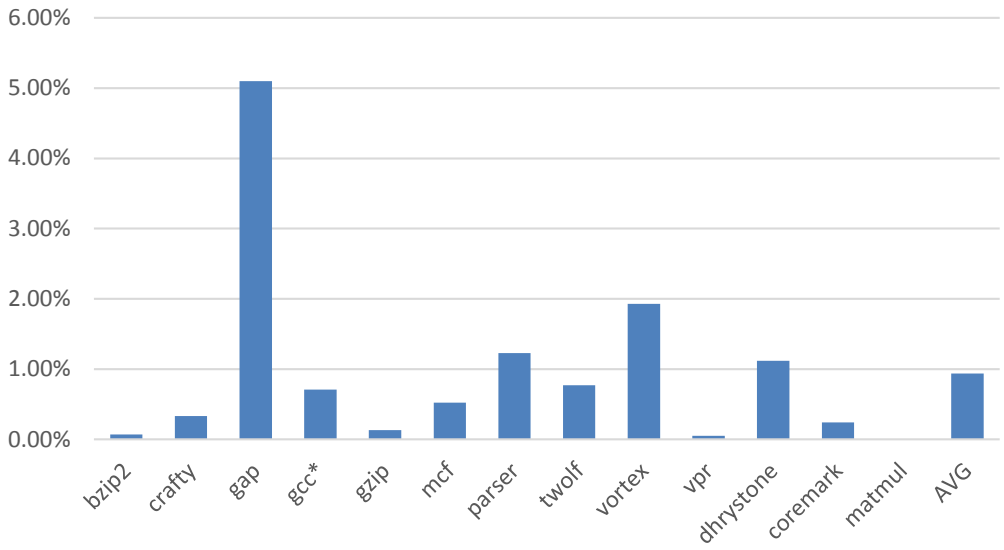


Figure 7.5: The runtime overhead measured with our implementation.

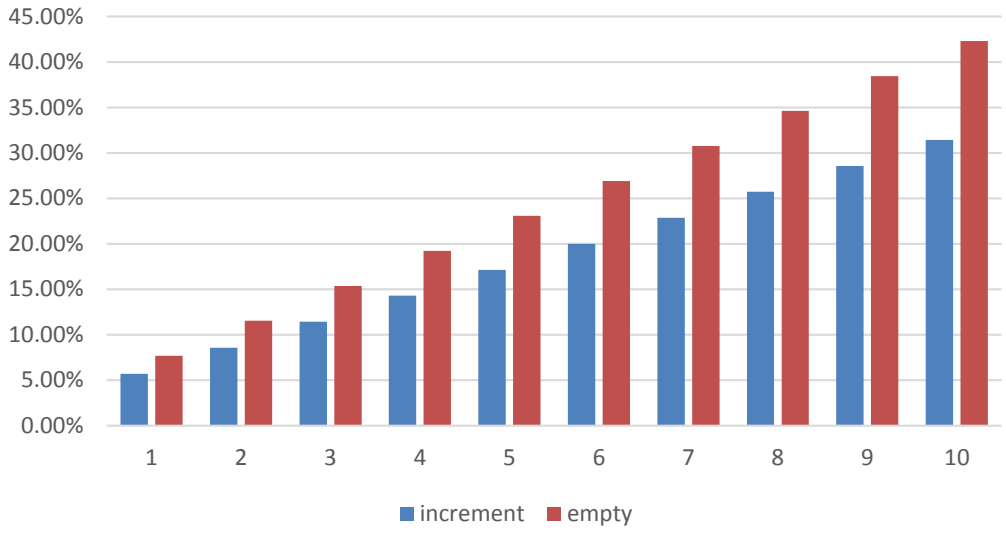


Figure 7.6: The runtime overhead added by using 1-10 labels on an empty function or a function that increments a value.

7.6.1 Runtime Overhead

When instrumenting only calls (both direct and indirect) and returns, the average overhead lies at a little under 1% as shown in Figure 7.5. In the case of `gap` benchmark, the reported overhead is the result of a tight loop executing a multitude of indirect calls to relatively small functions.

We also run two series of micro-benchmarks to see the effect of adding multiple labels to a function. We did this by executing a tight loop with an indirect call to one of two functions. The first was an empty function, which results in three assembly instructions. The second was a function that incremented a global variable, this has a body of ten instructions. We added one to ten labels on the function entry points. With these benchmarks we can find the maximum percentage of runtime overhead imposed when a function is called indirectly with CFI instrumentation. We present our results in Figure 7.6. In our previous design the maximum runtime overhead that could be imposed is the same as the overhead reported for the empty function with only one label. The runtime overhead is relative to the number of indirect call sites that can point to each function (i.e. the number of labels in the entry point) and the number of instructions in the function. In large functions, CFI instructions will account for a small percentage of the function's code. Thus, we expect that the performance overhead will be significantly less in real world applications. By also instrumenting indirect jumps, the overhead can increase; even though this depends on the total number of indirect branches that the program uses. For example, forward-edge protection in the jump table implementation of switch statements, can be accomplished

with the execution of just two additional instructions. In our new design, the granularity of forward-edge can be adjusted, i.e., use the same labels in some indirect call sites in order to reduce the number of labels in function entry points. Thus, application designers can opt to reduce forward-edge granularity in order to favor performance.

7.6.2 Hardware Overhead

We implemented our design initially without `longjmp` support and the recursion optimization. The resulting area overhead, as detailed by the reports of the Xilinx tools used to synthesize the design, is very low, using an additional 0.65% registers and 0.81% LUTs (look-up tables). The area overhead increases significantly to 2.52% registers and 2.55% LUTs, when placing the `longjmp` support and the recursion optimization.

7.6.3 Power Consumption

We measure the power consumption of our design using the Xilinx XPower Analyzer tool. For the unmodified design the tool reported 6072.11 mW power consumption. The required modifications for the CFI instructions increase the power consumption to 6149.32 mW. The full fledged design with CFI and SJ/LJ support has a power consumption of 6176.92 mW. The results indicate that the power consumption overhead is about 1.2%, which increases to 1.7% when adding `longjump` support.

7.7 Designing CFI in RISC-V Architecture

While, HCFI is a promising strategy for CFI since it offers enhanced granularity, the design requires radical changes in order to be applied in real-world general purpose architectures. In this section we will discuss each one of the design choices proposed in order to upgrade our academic work to an industry level architectural extension. Our design is at the time of writing this dissertation a mature proposal for the Shadow Stack and Landing Pads (SSLP) extension for RISC-V architecture.

7.7.1 Architecturally protected Shadow Stack

In our academic design we implemented the shadow stack as an in-core isolated memory area that can be only accessed through the shadow stack instructions. In-core memory is a very effective implementation both for security (since only shadow stack instructions can access the stack) and performance (in-core memory has substantially less access time). However, it is not a scalable solution (i.e., multiple threads), since it requires complete flush and repopulation of the shadow stack, on every context switch. This would significantly slow down the performance of context switching and subsequently the overall per-

formance of the system.

In SSLP, we introduced new encodings for memory page access rights in order to define shadow stack memory pages, i.e., shadow stacks are protected through page table attributes. The access rights for the shadow stacks ensure that, the pages can be only modified through shadow stack control instructions. Regular store operations as well as instruction fetching will result in an access fault. In a similar manner, shadow stack instructions that target non-shadow stack pages will also result in access faults. In this proposal we aim to offer CFI for each privilege level of the processor i.e., CFI protection in enabled from machine mode up to user mode. This is contrary to our previous design that only offered protection for the highest privilege level and targeted embedded devices.

Machine mode Shadow Stack

The machine mode for RISC-V is the most privileged execution level. This privilege level is responsible for running the firmware of the processor as well as secure execution environments. Machine mode is also responsible for setting up the environment for least privilege modes (e.g., Supervisor). Machine mode operates directly on physical memory and thus we cannot deploy traditional page access rights to define shadow stack pages. To achieve this we rely on Physical Memory Extension (PMP) RISC-V specification [158]. PMP defines a finite number of address range regions with individual access rights. The regions can be locked after initialisation until the next reset in order to prevent attackers from reconfiguring the access rights (e.g., through Code Reuse attacks).

In SSLP we deploy PMP, in order to define a region for shadow stacks. An issue with PMP is that there are no available bits for encoding the access rights for a shadow stack (i.e., Shadow Stack bit). To remedy this we reserved the encoding of non-readable, non-executable, writable (i.e., R=0, W=1, X=0) as a shadow stack region. This encoding does not make sense for any other purpose and thus can be safely used in our case. Finally, the shadow stack region for the machine mode is inaccessible for Supervisor and User mode.

Supervisor mode Shadow Stack

The supervisor mode is responsible for initializing the Operating System. Usually Virtual Addressing is available at this privilege level and we again use a similar definition for the shadow stack pages as with machine mode i.e., R=0, W=1, X=0.

User mode Shadow Stack

For the user mode, the allocation of shadow stack pages is the responsibility of the Operating System. During the process initialization, if the ELF contains the backwards-edge CFI flag, the operating system will allocate a shadow stack page and set the initial value

for the shadow stack pointer register. The shadow stack is enabled specifically for each application.

7.7.2 Instruction Extension

For designing the ISA extensions for RISC-V we had to take several factors into account in order to preserve compliance with the ABI, backwards compatibility and also support corner cases that we did not allow in our previous design.

A major issue we had to address, was the appropriate encoding space for these instructions. In our academic design we implemented CFI instructions in a new encoding space i.e., CFI type. This design however is not applicable to RISC-V since applications with CFI would result in executing illegal instructions if a processor does not have CFI support. A suitable encoding space in order to avoid such scenario are instructions that are treated as No Operation (NOPS) and not as illegal. In RISC-V there exists the *hint* encoding space which does exactly that, i.e., hint instructions do not interfere with the micro-architectural state in any way.

However, implementing the CFI instructions as *hints* would violate the definition of *hint* instructions and thus this idea was again not suitable. Eventually, the appropriate encoding space for implementing the CFI instructions is *zimops*. This encoding space is defined as *Maybe Operations* and are treated as *NOPs* if no functionality is implemented. The specification for *zimops* is not yet standard for RISC-V architecture, however the plan is to be ratified by Q2 '23.

Forward-edge CFI

For supporting the forward-edge control-flow integrity we designed label set and label check instructions, tailored to RISC-V architecture. In contrast to SPARC V8 architecture where we could encode 18-bit labels in each label set instruction, only 9 bits are available for immediates in *zimops* instructions. In order to support a large number of labels, especially in the case of per-indirect-call site labels, we designed three similar instructions that each set a different portion (low, mid, upper) of the label bits. With this approach we are able to support up to 25-bit wide labels. Respectively, there are three different label check instructions, one for each portion of the label. The label check procedure always begins from the lower 9-bits and it is the only valid target for indirect jumps. This ensures that during the function entry, all of the label bits will be checked. Another differentiation from HCFI is that the expected landing pad state is set by the *jalr* (Jump and Link Register) instruction. For functions that carry multiple labels, the compiler is responsible for generating code that will point to the appropriate sequence of label check instructions.

To summarize, the instructions and modifications for forward-edge CFI are the following:

- `lpsll`: Sets the 9 LS bits of the expected label.
- `lpmll`: Sets the bits between 8 and 15 of the expected label.
- `lpull`: Sets the bits between 16 and 23 of the expected label.
- `jalr`: Sets the state of the processor to expected landing pad. In this state only *lpcll* instructions are valid for execution.
- `lpcll`: Checks the 9 LS bits of the label. If the bits are equal the expected landing pad state is cleared. If the bits are not equal, the execution will result in an illegal instruction exception
- `lpcml`: Checks the label bits between 8 and 15. If the bits are not equal, the execution will result in an illegal instruction exception.
- `lpcul`: Checks the label bits between 16 and 23. If the bits are not equal, the execution will result in an illegal instruction exception.

Backwards-edge CFI

The backwards-edge CFI is enforced through a shadow stack in a similar manner as we did in HCFI. Compared to HCFI, in SSLP the shadow stacks are stored in main memory in order for our design to be scalable for multithreaded environments. Since there are no delayed branches in RISC-V architecture the copies of the return addresses are stored during function entry (in HCFI the copies were stored during the delay slot of indirect jumps). Another, challenging design choice for RISC-V architecture was the reduction of the code overhead of CFI instructions. Thus, it was decided that it was preferable to introduce a pop and check instruction, which is not compatible to the definition of RISC architectures (i.e., an instruction that loads from memory, increments a pointer and also utilizes comparison) but will be able to make use of the 16-bit instruction encoding (i.e., compressed ISA) and thus significantly reduce the CFI instruction footprint in protected binaries.

For supporting `setjump/longjump` the respective library functions (e.g., libc) are modified in order to read the current shadow stack address during setjump and storing it in the longjump context. Upon longjump, the shadow stack is unwinded to the address, stored in the setjump struct.

Finally, we introduced an additional shadow stack instruction that can be used by applications that use user-level threads (i.e., threads are not managed by the Operating System but within the user-level process). For example Goroutines in Go language [78] are lightweight threads that are managed directly by the Golang runtime in order to avoid the cost of switching to the supervisor level.

To summarize, the instructions for backward-edge CFI are the following:

- `sspush`: Stores the return address (either
  x1 or
  x5) in the address pointed by the shadow stack pointer and decrements the shadow stack pointer by either 4 or 8 bytes (depending on the architecture).

- `sspop`: Loads the shadow copy of the return address and compares it with the return address in the stack (usually register
  x1). If the shadow stack copy is different than the stack value an illegal instruction exception is raised. Finally, it increments the shadow stack pointer by 4 or 8 bytes.

- `ssamoswap src, addr, dst`: Atomically stores the value *addr* in *src* and loads the previous value in *dst*.

7.7.3 CFI Context Specific Registers

Several CSRs are required in order to implement hardware CFI extension. These registers are required for vital functions and ease of leverage by applications. For forward-edge we defined the label register and the expected landing pad state. The label CSR is a supervisor read-write register that holds the expected label of indirect branch targets. The Expected Landing pad is an architectural state register that is set during indirect jump instructions if the process has forward-edge CFI enabled. Additionally, Supervisor Previous Expected Landing Pad and Machine Previous Expected Landing Pad, are defined in order to ensure that if a trap is taken immediately after an indirect jump, the CFI state will be preserved when the process continues its execution. For backwards-edge CFI we defined the Shadow Stack Pointer CSR that hold the address of the current shadow stack. This CSR is an unprivileged read-write register.

Finally, we reserve fields in environment configuration and status registers in order to declare the availability of CFI in each privilege level and track the status of CFI (e.g., expected landing pad state). Specifically, in Machine Environment Configuration register we reserve bits that declare if CFI is enabled for use by supervisor mode. In Machine status register we reserve two bits that declare if backwards-edge and forward-edge CFI are available for user mode, as well as two bits for the previous expected landing pad state. This two bits are required in order to identify if during an interrupt (to supervisor or machine mode), the interrupted application was executing an indirect branch and was expending to check the validity of labels in the next issued instructions.

7.8 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 strategies proposed in the literature and the industry as well as their limitations.

7.8.1 Active Set Control Flow Integrity

Davi et al. [61] proposed HAFIX, a system for protecting backward edges based on active set CFI. HAFIX is based on Active Set Control Flow Integrity, i.e. only functions that have been called are available for return targets. HAFIX deploys 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. A return can only point to an `active` function. However, the aforementioned design has the disadvantage of allowing the attacker to jump to any active function [189]. This is important, since an attacker can use stack unwinding, to avoid the execution of CFIDEL instructions and eventually mark every function as active, effectively permitting jumps anywhere in the program. In our design we use an architecturally protected shadow stack, a technique considered to be the state of the art for protecting backward edges. Moreover, our design offers forward edge protection. HAFIX proposes the use of software techniques for protecting forward-edges.

7.8.2 Pointer Integrity (Cryptographically enforced CFI)

ARM presented Pointer Authentication Code (PAC) [115]. This mechanism utilizes cryptographic primitives (hashing) in order to verify that the control flow pointers are not corrupted before using them. This mechanism utilizes cryptographic primitives: a keyed message authentication code (mac) with a modifier. The modifier includes process context, and in some instructions the stack pointer, in order to verify that the control flow pointers are not corrupted before using them. The pointer authentication code (PAC) of each control flow pointer is stored in the unused bits of the pointer. The number of otherwise unused bits varies based on the size of the virtual address space. There are no free bits when a 32-bit VA is used, 24 free bits in 40-bit VA, 16 in 24-bit VA and 9 when 59 bit VA is used. Since there are fewer free bits with the larger VAs (beyond 32-bit), the strength of the MAC entropy weakens as the VA size increases. PACs are calculated and authenticated using custom instructions. These instructions have opcodes that are NOP on processors that do not support PAC.

- PAC Pointer, &Pointer (usually in stack) : calculates a MAC using a process key
- AUTH Pointer, &Ppointer : authenticates the MAC using the process key

Each process has a unique modifier. Typically there is a set of static keys established at boot time for code and data. Then each context has a context modifier which is mixed

into the hash. The modifier is used in order to calculate and authenticate the control flow pointers. This technique can protect both data and control pointers, however it requires significant chip area and has non negligible runtime overhead. Moreover, it is not suitable for 32-bit processors. Reserving unused address bits conflicts with several other extensions - e.g. J-extension (pointer masking), future memory tagging extensions, etc. Finally, there is an increase in power consumption due to the additional crypto operations.

A recent study from Googles project zero identified several vulnerabilities in this technology [84]. Moreover, it has been proven that micro-architectural side-channels can be used in order to bypass this scheme [157]. Our design is not affected by micro-architectural side-channels, since we do not rely on hiding any of the Control Flow metadata (e.g., labels, pointer addresses, etc.). Thus, pointer authentication cannot offer similar levels of protection with our design. Finally, the use of cryptographic primitives in PAC instructions imposes significantly more overhead in terms of performance and area compared to our design.

Recent revisions of ARM PAC are complemented with Branch Target Identification [20] instructions in order to guard against execution of instructions that are not intended as branch targets.

7.8.3 Intel Control Flow Enforcement Technology

In June 2016, Intel announced Control-flow Enforcement Technology [8] (CET). In CET a shadow stack is defined in order to protect backward-edge control flow transfers in a manner similar 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. This is contrary to our design, since the same instruction `jalr` is used for both indirect call and return and would also violate the compliance with RISC. 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.

7.8.4 Memory Tagging

Recently ARM introduced memory tagging in recent processors [147]. The key idea of this mechanism is that each 16-byte block of memory will be tagged using a 4-bit value. Each pointer will hold the tag of the valid memory blocks which can be accessed on its 4 most-significant bits. Memory operations are only valid if the address and the target have the same tag. Memory blocks are re-tagged when freed. This mechanism can prevent a lot of exploitation techniques which rely on memory corruption vulnerabilities. For example, a pointer pointing to a buffer will not be able to access adjacent memory blocks beyond its buffer bounds, effectively protecting adjacent memory from being overwritten if the buffer overflows. A potential problem with this mechanism is that 4 bit tags will offer reduced entropy and thus many memory blocks will have the same tag. The performance overhead of this mechanism has not yet been measured. If the performance overhead is substantial, memory tagging will not be a practical solution. In a similar manner with PAC this technique requires that the MS bits of the VA are used for storing the tag. Thus, this technique cannot be deployed in 32-bit architectures. In terms of memory overhead this technique imposes 3.25% memory for storing the tags. Moreover there is noticeable memory fragmentation due to the need of 16 byte alignment. Memory tagging for stack locations - to address pointers passed through the stack - could have additional overheads as the stack pointer cannot have a static tag and no checking could lead to imprecisions.

7.8.5 Dynamic Information Flow Tracking

Finally, a notable technique proposed in the literature, in order to counter memory corruption related exploits, is Dynamic Information Flow Tracking (DIFT) [184]. The key concept of this mechanism is to taint memory regions where untrusted data are residing, and track their propagation in the application's address space. The data input sources of an application (e.g., network, user interface) are tagged with a label that *taint* the memory region the data reside. Any new data resulting from computation or memory operation with tainted data as source, also become tainted. Exploits are detected with a predefined policy, depending on the implementation of DIFT. In the general scenario, 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 where 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 [26, 46], the protected application is oblivious of the mechanism, thus there is no need for source code modifications. While very effective, this mechanism requires each word in memory to have associated metadata. For more complex processors with multiple levels of caches, OOO execution and multiple instructions retirement per cycle this mechanism could lead to impractical

overhead both in terms of runtime performance and circuit are.

Table 7.2: Comparison of CFI strategies for preventing Control-Flow hijacking attacks.

Technique	Forward Edge	Backward Edge	Memory Overhead	Runtime Overhead	Architectural Modifications
Pointer Authentication [115]	Entropy based	Entropy based	Minor	Moderate	Major
Active Set CFI [61]	None	Coarse	Minor	Minor	Minor
DIFT [46]	Fine grained	Fine grained	Significant	Significant	Major
Memory Tagging [147]	Entropy based	Entropy based	Modest	Minor	Major
CET [8]	Borderline	Fine	Minor	Minor	Modest
Shadow Stack & Labels	Complete	Fine	Minor	Minor	Modest

7.9 Summary

In this chapter, we presented the design, implementation and evaluation of a flexible and policy-agnostic Control-Flow Integrity Instruction Set Extension. Our extensions introduced less than 1% runtime overhead on average and less than 2% increase in power consumption, while only imposing very little overhead in terms of additional hardware circuitry (less than 2.55%). We finally presented how the lessons learned through this work can be applied in order to form an industry ready CFI Extension design.

Chapter 8
Future work and Conclusion

Computing system have expanded to nearly every aspect of modern societies. Internet connected devices are responsible for the operation of vehicles (Smart cars), factories (SCADA), Cities, *etc.*. Thus, it is important to ensure that these systems cannot be disrupted for malicious intent. Through the years a lot of mechanisms and strategies have been designed, implemented and deployed in order to offer enhanced security guarantees to their use cases. However, security is not cheap and causes proportionate runtime performance overhead as well as increased power consumption. As we presented in this dissertation more often than not, very effective mechanisms are eventually abandoned due to performance impact and the limitations they introduce.

In this dissertation we try to address these issues by pushing security mechanisms in the hardware level. This strategy is a well-trodden path for various aspects of computing systems (i.e., accelerate specific operations by designing special circuity). However, this is by no means an easy task, since it requires careful and precise design of the security mechanism aspects that are going to be implemented in the hardware level. Contrary, to software based mechanisms, hardware is immutable and thus any changes will only apply after newer generations of the hardware are introduced. As we presented in the 3 there are examples of hardware assisted mechanisms that ended up being abandoned by developers due to substantial performance impact, limitations and ineffectiveness. We explored two avenues of leveraging hardware for security, the first is the utilization of hardware extensions already present in modern commercial of the self processors and the second is the clean-slate design of architecturally assisted mechanisms. The mechanisms and strategies we presented cover a wide range of threats with the common denominator being attacks on memory.

8.1 Synopsis of Contributions

The main contributions of this dissertation are as follows:

- The implementation of a main memory encryption scheme, able to run on off-the-

self hardware and support legacy applications. Our design leverages AES-NI [98] and IOMMU [11, 12] in order to protect systems against physical attacks. We instrumented applications using Intel's dynamic instrumentation tool called PIN [27], which provides the run-time environment, and supports legacy applications without any code modification. With our approach, application data are always encrypted in main memory, using a 128-bit AES key. We also experimentally quantify the cost of keeping sensitive data secure in practical, real-world scenarios in terms of runtime performance.

- The implementation of a hybrid language-binary framework for protecting against the few native add-ons present in modern Node.js applications. We developed a fine-grained read-write permission model applied at the boundaries of native add-ons, offering a unified view and isolation of privilege cutting across the barrier between the language wrapper and the binary core. To achieve this we leverage the protection keys hardware feature [99], in order to isolate the execution of untrusted libraries. We experimentally evaluated our design against real-world exploits and the runtime performance using a plethora of libraries and real-world applications.

- The design and implementation of hardware-assisted AES Instruction Set Randomization. ISR using AES requires significant modifications on the processor architecture, however, offers enhanced protection against key guessing attacks and cryptanalysis. We demonstrated the feasibility of our approach by presenting the prototype implementation of AESASIST by modifying the Leon3 SPARC V8 processor [4], a 32-bit open-source synthesizable processor [68]. Our experimental evaluation results prove that our design is able to prevent code injection attacks as well as the majority of code reuse attacks with acceptable overhead.

- The design and implementation of hardware-assisted, policy agnostic CFI. Our design offers increased granularity (especially in forward-edge situations);and covers a couple of intrinsic situations (including the instrumentation of fall-through functions and indirect jumps, such as `switch` statements, within functions. Performance-wise, we proved that the implementation in hardware is the optimal choice. Our design, imposes minimal overhead, both in terms of runtime performance and power consumption. Finally, we presented how our research efforts on CFI formed the design of the CFI specification for RISC-V architecture.

- We presented a short summary of the most prominent hardware assisted security mechanisms that formed many aspects of today's computing systems through the years. This summary is a strong proof that hardware assisted security mechanisms are not only increasing in absolute numbers but also design complexity.

8.2 Directions for Future Work and Research

There are several research directions that can be derived through this dissertation:

Novel architectural assisted security mechanisms.

Security threats are continuously evolving in parallel with computing capabilities. It is a common belief that total and verifiable security is nearly impossible to achieve (while preserving the system's functionality to a practical level). Thus, as new threats are emerging academia and industry will introduce even more strategies in an effort to thwart them. There are two avenues for the development of hardware-assisted security mechanisms. Directly tackle the targeted threat through the clean-slate design of architectural extensions. Or, accelerate operations and primitives of proven software security approaches (e.g., CFI was initially proposed as a software only approach).

Processor type specific security mechanism designs.

In this dissertation we designed hardware assisted security mechanisms that are general purpose and do not take into account use-cases with specific constrains. Future designs can be of variant complexity depending on the processor type e.g., ultra-low power processors with constrains regarding extensions must be complemented with light-weight variances of security designs. Taking into account these constrains during the design process, deciding on the best approach and evaluating the various key performance indexes (security, power consumption, area overhead, performance impact, *etc.*) requires a lot of research.

Utilization of hardware features for security.

We presented how we leverage hardware features already available in commodity processors for security strategies. The ever increasing number of architectural extensions creates the opportunity to research how these can be leveraged for solving security related issues beyond the original threat model. To give an example, at the time this dissertation was written we were evaluating a design that leverages Instruction Set Randomization in order to *hide* the keys of cryptographic libraries. This can apply even to extensions that were not designed for security at all.

Adaptive security designs in the era of reconfigurable hardware.

In an effort to offer even more computing capabilities to customers, major cloud infrastructure providers introduced reconfigurable computing solutions i.e., Field Programmable Gateway Arrays (FPGAS) [155, 198] in their infrastructures. The ample heterogeneity of

these architectures is beneficial for flexibility, performance and power consumption. However, given that cloud infrastructures are by multi-tenant environments the security is of utmost importance in order to assure that threat actors will not interfere with benign users, Thus, securing the threat landscape of FPGAs can only be achieved through the design of hardware-assisted security mechanisms. Additionally, since FPGAs can be rapidly reconfigured with new hardware designs, the strategies can be rapidly redesigned to address issues and adapt to emerging threats. Finally, we believe that reconfigurable logic will eventually reach commercial computing systems e.g., personal computers, smart phones, *etc.*. Small steps have already been taken towards this direction, Apple's Iphone 7 includes an small FPGA for Digital Signal Processing [106]. This does not only introduce new attack surfaces that can be researched and documented but also opportunities to design new mechanisms that benefit from the flexibility of reconfigurable hardware.

8.3 Conclusion

This dissertation examines how hardware can assist security mechanisms targeting the modern threat landscape, by leveraging existing architectural features as well as designing new ones. Our work demonstrated that hardware can offer enhanced security and better runtime performance compared to similar software only solutions. We focused on how we can thwart memory related exploitation techniques both physical and remote.

We started by leveraging common hardware extensions in commodity processors in order to prevent attackers with physical access but no authority on a machine, from extracting sensitive data. We designed our mechanism to be transparent for the applications that use it. We then proposed different schemes of operation in an effort to minimize the runtime performance impact of our design. In a similar manner, we leveraged architectural extensions in order to enforce isolated execution for third-party untrusted libraries in managed runtime environments. Our results prove our notion that by leveraging hardware, we are able to keep performance impact within practical limits.

We additionally explored the design and implementation of our own hardware extensions for assisting security mechanisms aiming to prevent Code Reuse and Code Injection attacks in presence of memory corruption vulnerabilities. We designed hardware components that would accelerate the various operations and checks required by each security strategy. Our evaluation proved that our designs were more secure while imposing significantly less runtime performance impact.

In summary, this dissertation demonstrates how hardware can be leveraged in order to effectively secure systems against modern sophisticated threats while preserving normal functionality and imposing the least possible runtime performance impact. We presented the decisions we made during the design phase of each mechanism we worked on in order to achieve our goals i.e., security, performance, compatibility. The trend of including

more and more security related extensions in commercial CPUs is a concrete proof for the correctness of the approaches on security this dissertation follows. We hope that the design decision processes, principles and comparative research provided in this work, will be applied on designing effective and practical architectural solutions for current and future threats.

Bibliography

[1] ARM trustzone. https://www.arm.com/products/security-on-arm/trustzone.

[2] No Execute bit. https://en.wikipedia.org/wiki/NX_bit.

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

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

[5] Linux Kernel Remote Buffer Overflow Vulnerabilities. http://secwatch.org/advisories/1013445/, 2006.

[6] OpenBSD IPv6 mbuf Remote Kernel Buffer Overflow. http://www.securityfocus.com/archive/1/462728/30/0/threaded, 2007.

[7] Microsoft Windows TCP/IP IGMP MLD Remote Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/27100, 2008.

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

[9] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[10] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[11] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel virtualization technology for directed i/o. *Intel technology journal*, 10(3).

[12] Advanced Micro Devices Inc. AMD I/O Virtualization Technology (IOMMU). http://support.amd.com/TechDocs/48882_IOMMU.pdf.

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

[14] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. Jsand: Complete client-side sandboxing of third-party javascript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 1–10, New York, NY, USA, 2012. ACM.

[15] AMD. AMD EPYC Hardware Memory Encryption. https://developer.amd.com/sev/#:~:text=AMD%20EPYC%20Hardware%20Memory%20Encryption&text=AES%2D128%20encryption%20engine%20embedded,key%20generation%20and%20key%20management.

[16] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent 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*, volume 13, 2013.

[17] Starr Andersen and Vincent 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.

[18] ARM. Memory Tagging Extension: Enhancing memory safety through architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety.

[19] ARM. Arm memory domains, 2018.

[20] ARM. Branch target identification, 2021.

[21] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[22] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The devil is in the constants: Bypassing defenses in browser jit engines. In *NDSS*. The Internet Society, 2015.

[23] Todd M. Austin, Scott E. Breach, and Gurindar 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*, PLDI '94, pages 290–301, New York, NY, USA, 1994. ACM.

[24] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. Randomized Instruction Set Emulation. *ACM Transactions on Information and System Security*, 8(1), 2005.

[25] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[26] Iulia Bastys, Frank Piessens, and Andrei Sabelfeld. Prudent design principles for information flow control. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, pages 17–23, 2018.

[27] Sion Berkowits. Pin - a dynamic binary instrumentation tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`, 2012.

[28] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen, and Pierre Wilke. Compiling sandboxes: Formally verified software fault isolation. In *European Symposium on Programming*, pages 499–524. Springer, Cham, 2019.

[29] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Board Range of Memory Error Exploits. In *USENIX Security Symposium*, 2003.

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

[31] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 227–242. IEEE, 2014.

[32] Erik-Oliver Blass and William Robertson. Tresor-hunt: attacking cpu-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 71–78, 2012.

[33] Dion Blazakis. Interpreter exploitation: Pointer inference and jit spraying. *BlackHat DC*, 2010.

[34] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Sympo-*

sium on Information, Computer and Communications Security, pages 30–40. ACM, 2011.

[35] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*, pages 30–40, 2011.

[36] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proc. of USENIX Summer*, pages 87–98, 1994.

[37] Stephen W. Boyd, Gaurav S. Kc, Michael E. Locasto, Angelos D. Keromytis, and Vassilis Prevelakis. On the General Applicability of Instruction-Set Randomization. *IEEE Transactions on Dependable Secure Computing*, 7(3), 2010.

[38] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[39] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[40] Bugtraq. Getting around non-executable stack (and fix). `https://seclists.org/bugtraq/1997/Aug/63`.

[41] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, 2015.

[42] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, 2014.

[43] Ellick M Chan, Jeffrey C Carlyle, Francis M David, Reza Farivar, and Roy H Campbell. Bootjacker: compromising computers using forced restarts. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, pages 555–564. ACM, 2008.

[44] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *ACM Conference on Computer and Communications Security (CCS)*, pages 559–572, 2010.

[45] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems. In *Asia-Pacific Workshop on Systems (APSys)*, 2011.

[46] Kejun Chen, Xiaolong Guo, Qingxu Deng, and Yier Jin. Dynamic information flow tracking: Taxonomy, challenges, and opportunities. *Micromachines*, 12(8):898, 2021.

[47] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, 2005.

[48] S. Christey and A. Martin. Vulnerability Type Distributions in CVE. `http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf`, 2007.

[49] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: Hardware-Enforced Control-Flow Integrity. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, CODASPY '16, 2016.

[50] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the Twentieth International Conference on ASPLOS '15*.

[51] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. *Dependent Types for Low-Level Programming*, pages 520–535. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[52] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. {PKU} pitfalls: Attacks on pku-based memory isolation systems. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1409–1426, 2020.

[53] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963. ACM, 2015.

[54] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, volume 98, pages 63–78, 1998.

[55] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *USENIX Security Symposium*, 2001.

[56] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointguardTM: Protecting Pointers from Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*, 2003.

[57] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[58] D. R. Piegdon. Hacking in physically addressable memory: a proof of concept. `http://eh2008.koeln.ccc.de/fahrplan/attachments/1067_SEAT1394-svn-r432-paper.pdf`.

[59] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Real-World Buffer Overflow Protection for Userspace & Kernelspace. In *USENIX Security Symposium*, 2008.

[60] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security, ASIACCS*, volume 15, 2015.

[61] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. Hafix: hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference*, page 74. ACM, 2015.

[62] Willem De Groef, Fabio Massacci, and Frank Piessens. Nodesentry: Least-privilege library integration for server-side javascript. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 446–455, New York, NY, USA, 2014. ACM.

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

[64] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve 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*, ASPLOS XIII, pages 103–114, New York, NY, USA, 2008. ACM.

[65] John A Dilley. *Web server workload characterization*. Hewlett-Packard Laboratories, Technical Publications Department.

[66] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[67] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, New York, NY, USA, 2015. ACM.

[68] Gaisler Research. Leon3 synthesizable processor. `http://www.gaisler.com`.

[69] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, et al. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 469–484. ACM, 2019.

[70] GCC. Remove MPX support. `https://gcc.gnu.org/ml/gcc-patches/2018-04/msg01225.html`.

[71] George Christou. ASIST Leon3 source code. `https://github.com/G3org1o/grlib-asistmmu-aes`.

[72] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *USENIX Security Symposium*, 2012.

[73] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 575–589. IEEE, 2014.

[74] Enes Göktaş, Angelos Economopoulos, Robert Gawlik, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. Bypassing clang's safestack for fun and profit. *Black Hat Europe*, page 21, 2016.

[75] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 227–242. IEEE, 2018.

[76] Google. Orinoco: young generation garbage collection, 2017.

[77] Google. V8 garbage collector, 2018.

[78] Google. Go language goroutines, 2021.

[79] Google. Ignition interpreter, 2022.

[80] Google. Sparkplug javascript compiler, 2022.

[81] Google. V8 javascript engine, 2022.

[82] Google. V8's public api, 2022.

[83] Google Project Zero. Examining Pointer Authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[84] Google Project Zero. Examining Pointer Authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[85] G.Ou. Cryogenically frozen ram bypasses all disk encryption methods. . http://www.zdnet.com/article/cryogenically-frozen-ram-bypasses-all-diskencryption-methods/.

[86] Michael Gruhn and Tilo Müller. On the practicability of cold boot attacks. In *Proceedings of the 2013 International Conference on ARES '13*.

[87] Michael Gruhn and Tilo Müller. On the practicability of cold boot attacks. In *2013 International Conference on Availability, Reliability and Security*, pages 390–397. IEEE, 2013.

[88] Shay Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.

[89] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[90] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5), May 2009.

[91] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 489–504, 2019.

[92] Michael Henson and Stephen Taylor. Memory encryption: A survey of existing techniques. *ACM Comput. Surv.*, 46(4), March.

[93] Graydon Hoare. RUST programming language. `https://www.rust-lang.org/`.

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

[95] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and Practical Defense Against Code-Injection Attacks using Software Dynamic Translation. In *ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2006.

[96] IBM. Kernel storage protection keys, 2022.

[97] Intel. Ia64 software development manual, 2000.

[98] Intel. Intel AES-NI, 2012. `https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html`.

[99] Intel. Intel Memory Protection Keys. `https://www.kernel.org/doc/html/latest/core-api/protection-keys.html`, 2022.

[100] Intel Xeon Processor Intel. Intel architecture memory encryption technologies specification, 2019.

[101] Intel Corporation. Software guard extensions programming reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.

[102] Xuxian Jiang, Helen J. Wangz, Dongyan Xu, and Yi-Min Wang. RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2007.

[103] kashif. node-cuda provides nvidia cuda™ bindings for node.js., 2022.

[104] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[105] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection Against Return-to-User Attacks. In *USENIX Security Symposium*, 2012.

[106] FPGA key. Smartphone fpga, 2020.

[107] keyhash. Cryptonight hashing functions for node.js., 2022.

[108] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 132–148, 2022.

[109] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[110] Tim Kornau. Return oriented programming for the arm architecture. *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[111] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *USENIX OSDI*, 2014.

[112] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. 2018.

[113] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. 2017.

[114] Kevin P Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996.

[115] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. Pac it up: Towards pointer integrity using arm pointer authentication. In *28th USENIX Security Symposium*, 2019.

[116] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *ACM SIGARCH Computer Architecture News*, volume 41, 2013.

[117] Linux. x86: remove Intel MPX. `http://lkml.iu.edu/hypermail/linux/kernel/1812.0/04478.html`.

[118] Linux. Secure Computing. `https://lwn.net/Articles/656307/`, 2022.

[119] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[120] LLVM. Clang documentation on SafeStack. `https://clang.llvm.org/docs/SafeStack.html`.

[121] Jonas Magazinius, Daniel Hedin, and Andrei Sabelfeld. Architectures for inlining security monitors in web applications. In *International Symposium on Engineering Secure Software and Systems*, pages 141–160. Springer, 2014.

[122] Alex Markuze, Adam Morrison, and Dan Tsafrir. True iommu protection from dma attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16.

[123] Alex Markuze, Adam Morrison, and Dan Tsafrir. True iommu protection from dma attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 249–262, 2016.

[124] Patrick McGregor, Tim Hollebeek, Alex Volynkin, and Matthew White. Braving the cold: New methods for preventing cold boot attacks on encryption keys. In Black Hat Security Conference 2008.

[125] Patrick McGregor, Tim Hollebeek, Alex Volynkin, and Matthew White. Braving the cold: New methods for preventing cold boot attacks on encryption keys. In *Black Hat Security Conference*, 2008.

[126] MITRE. Cve-2020-28248, 2020.

[127] Robert Morris and Ken Thompson. Password security: A case history. *Commun. ACM*.

[128] Tilo Müller, Andreas Dewald, and Felix C. Freiling. Aesse: A cold-boot resistant implementation of aes. In *Proceedings of the Third European Workshop on System Security*, EUROSEC '10.

[129] Tilo Müller, Felix C. Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11. USENIX Association, 2011.

[130] Vijay Nagarajan, Rajiv Gupta, and Arvind Krishnaswamy. Compiler-assisted memory encryption for embedded processors. In *Proceedings of the 2nd International Conference on HiPEAC'07*. Springer-Verlag.

[131] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 245–258, 2009.

[132] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *International Symposium on Memory Management (ISMM)*, pages 31–40, 2010.

[133] Nergal. The Advanced return-into-lib(c) Exploits: PaX Case Study. *Phrack*, 11(58), 2001.

[134] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[135] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747, 2012.

[136] Node.js. Native abstractions for node.js. 2022.

[137] nodejs. What is node-api?, 2022.

[138] ohmu. The missing posix system calls for node., 2022.

[139] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):28, 2018.

[140] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58, 2010.

[141] Aleph One. Smashing The Stack For Fun And Profit. *Phrack Magazine*, 7(49), 1996.

[142] openJS Foundation. node.js, 2009.

[143] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06. Springer-Verlag, 2006.

[144] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. Asist: architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[145] Vasilis Pappas, Michalis Polychronakis, and Angelos 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*, SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society.

[146] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, 2019.

[147] Aditi Partap and Dan Boneh. Memory tagging: A memory efficient design. *arXiv preprint arXiv:2209.00307*, 2022.

[148] Linda Dailey Paulson. New Chips Stop Buffer Overflow Attacks. *IEEE Computer*, 37(10), 2004.

[149] PaX Tream. Homepage of PaX. `http://pax.grsecurity.net/`.

[150] Phrack Magazine. BYPASSING STACKGUARD AND STACKSHIELD, 2000. `http://phrack.org/issues/56/5.html`.

[151] Georgios Portokalidis and Angelos D. Keromytis. Fast and Practical Instruction-Set Randomization for Commodity Systems. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[152] Prior99. Unofficial bindings for node to libpng., 2022.

[153] PrivateCore. Trustworthy computing for OpenStack with vCage. `http://privatecore.com/vcage/`.

[154] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*, pages 563–577, 2020.

[155] Andrew Putnam. Fpgas in the datacenter: Combining the worlds of hardware and software development. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 5–5, 2017.

[156] Tim Rains, Matt Miller, and David Weston. Exploitation trends: From potential risk to actual risk. In *RSA Conference*, 2015.

[157] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. Pacman: attacking arm pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 685–698, 2022.

[158] RISC-V. Physical memory protection, 2021.

[159] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[160] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, Dec 1996.

[161] Jonathan Salwan. Ropgadget tool, 2015.

[162] Sascha Schirra. Ropper.

[163] Theodoor Scholte, William Robertson, Davide Balzarotti, and Engin Kirda. An empirical analysis of input validation mechanisms in web applications and languages. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1419–1426, 2012.

[164] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for pku-based memory isolation systems. In *Proceedings of the 31th USENIX Security Symposium*, 2022.

[165] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys–efficient in-process isolation for risc-v and x86. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1677–1694, 2020.

[166] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[167] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and Reconfigurable Software Dynamic Translation. In *International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO)*, 2003.

[168] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[169] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[170] Patrick Simmons. Security through amnesia: A software-based solution to the cold boot attack on disk encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11.

[171] Kanad Sinha, Vasileios P Kemerlis, and Simha Sethumadhavan. Reviving instruction set randomization. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 21–28. IEEE, 2017.

[172] Sergei Skorobogatov. Low temperature data remanence in static ram. Technical report, University of Cambridge, Computer Laboratory, 2002.

[173] Asia Slowinska, Traian Stancescu, and Herbert Bos. Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation. In *USENIX Annual Technical Conference (ATC)*, 2012.

[174] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy*, 2013.

[175] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, May 2013.

[176] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[177] Snyk. Snyk vulnerability database, 2021.

[178] Snyk. node-sass vulnerabilities, 2022.

[179] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? the Effectiveness of Instruction Set Randomization. In *USENIX Security Symposium*, 2005.

[180] Standard Performance Evaluation Corporation (SPEC). SPEC CINT2000 Benchmarks. `http://www.spec.org/cpu2000/CINT2000`.

[181] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2012.

[182] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers 9*, pages 21–41. Springer, 2013.

[183] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03.

[184] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices*, 39(11):85–96, 2004.

[185] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*, pages 48–62, 2013.

[186] Gang Tan et al. *Principles and implementation techniques of software-based fault isolation*. Now Publishers, 2017.

[187] PaX Team. Pax address space layout randomization (aslr), 2003.

[188] Mike Ter Louw, Phu H Phung, Rohini Krishnamurti, and Venkat N Venkatakrishnan. Safescript: Javascript transformation for policy enforcement. In *Nordic Conference on Secure IT Systems*, pages 67–83. Springer, 2013.

[189] Michael Theodorides and David Wagner. Breaking active-set backward-edge cfi. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 85–89. IEEE, 2017.

[190] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(2), January 2010.

[191] Nathan Tuck, Brad Calder, and George Varghese. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2004.

[192] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. {ERIM}: Secure, efficient in-process isolation with protection keys ({MPK}). In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1221–1238, 2019.

[193] Nikos Vasilakis, Grigoris Ntousakis, Veit Heller, and Martin C. Rinard. Efficient module-level dynamic analysis for dynamic languages with module recontextualization. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1202–1213, New York, NY, USA, 2021. Association for Computing Machinery.

[194] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing dynamic library compromise on node.js via rwx-based privilege reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, New York, NY, USA, 2021. Association for Computing Machinery.

[195] Victor van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. Memory Errors: The Past, the Present, and the Future. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*, pages 86–106, 2012.

[196] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by) pass! practical, secure, and fast pku-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 266–282, 2022.

[197] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving Integer Security for Systems with KINT. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2012.

[198] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu. When fpga meets cloud: A first look at performance. *IEEE Transactions on Cloud Computing*, 10(2):1344–1357, 2020.

[199] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[200] Yoav Weiss and Elena Gabriela Barrantes. Known/Chosen Key Attacks against Software Instruction Set Randomization. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[201] Jos Wetzels. Hidden in snow, revealed in thaw: Cold boot attacks revisited. *arXiv preprint arXiv:1408.0725*, 2014.

[202] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, November 2016.

[203] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 457–468. IEEE, 2014.

[204] Xilinx. Xilinx University Program XUPV5-LX110T Development System. `http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf`, 2011.

[205] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *Usenix Security*, pages 337–352, 2013.

[206] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. Case: Cache-assisted secure execution on arm processors. In *Security and Privacy (SP), 2016 IEEE Symposium on*, S&P '16.

[207] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 995–1010, USA, 2019. USENIX Association.

Appendix A
Publications

Publications

The research activity related to this thesis has so far produced the following publications.

(1) No Sugar but all the Taste! Memory Encryption without Architectural Support. Panagiotis Papadopoulos, George Vasiliadis, George Christou, Evangelos Markatos, Sotiris Ioannidis. European Symposium on Research in Computer Security, 2017

(2) On Architectural Support for Instruction Set Randomization. George Christou, Giorgos Vasiliadis, Vassilis Papaefstathiou, Antonis Papadogiannakis, Sotiris Ioannidis. ACM Transactions on Architecture and Code Optimization (TACO), 2020

(3) Hard Edges: Hardware-Based Control-Flow Integrity for Embedded Devices. George Christou, Giorgos Vasiliadis, Elias Athanasopoulos, Sotiris Ioannidis. International Conference on Embedded Computer Systems, 2022

(4) BinWrap: Hybrid Protection Against Native Node. js Add-ons. George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P Kemerlis, Nikos Vasilakis. ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS), 2023