University of Crete
Computer Science Department

Architectural Support for Instruction Set Randomization

Laertis Loutsis
Master's Thesis

October 2014
Heraklion, Greece

University of Crete

Computer Science Department

Architectural Support for Instruction Set Randomization

Thesis submitted by

Laertis Loutsis

in partial fulfillment of the requirements for the

Master's of Science degree in Computer Science

THESIS APPROVAL

Author:	
	Laertis Loutsis
Committee approvals:	
	Evangelos P. Markatos
	Professor, Thesis Supervisor
	Sotiris Ioannidis
	Principal Researcher
	Maria Papadopouli
	Associate Professor
Departmental approval:	
	Antonis Argyros
	Professor, Director of Graduate Studies

Heraklion, October 2014

Abstract

Code injection attacks continue to pose a threat to today's computing systems, as they exploit software vulnerabilities to inject and execute arbitrary or malicious code. Instruction Set Randomization (ISR) is able to protect a system against remote machine code injection attacks by randomizing the instruction set of each process. This way, the attacker will inject invalid code that will fail to execute on the randomized processor and thus, the attack will fail as well. However, all the existing implementations of ISR are based on emulators and binary instrumentation tools that (i) incur a significant runtime performance overhead, (ii) limit the ease of deployment of ISR, (iii) cannot protect the underlying operating system kernel, and (iv) are vulnerable to evasion attempts trying to bypass ISR protection.

To address these issues we propose ASIST: an architecture with hardware and operating system support for ISR. We present the design and implementation of ASIST by modifying a SPARC processor, mapping it onto an FPGA board and finally running our modified Linux kernel to support the new features. The operating system loads the randomization key of each running process into a newly defined register, and the modified processor decodes the process's instructions with this key before execution. Moreover, ASIST protects the system against attacks that exploit kernel vulnerabilities to run arbitrary code with elevated privileges, by using a separate randomization key for the operating system.

We show that ASIST transparently protects all applications and the operating system kernel from machine code injection attacks with less than 1.5% runtime overhead, while only requiring 0.7% additional hardware.

Supervisor: Professor Evangelos Markatos

Περίληψη

Οι επιθέσεις εμβόλιμου κώδικα συνεχίζουν και αποτελούν απειλή για τα σημερινά υπολογιστικά συστήματα, εφόσον εκμεταλεύονται αδυναμίες του λογισμικού ώστε να καταφέρουν να εισβάλουν στο απομακρυσμένο σύστημα και να εκτελέσουν αμφίβολο ή κακόβουλο κώδικα. Η τεχνική της τυχαιοποίησης του συνόλου του εντολών του επεξεργαστή (ISR) έχει τη δυνατότητα να προστατέψει ένα σύστημα ενάντια σε απομακρυσμένες επιθέσεις εμβόλιμου κώδικα τυχαιοποιώντας το σύνολο των εντολών της κάθε διεργασίας. Με αυτόν τον τρόπο, ο εμβόλιμος κώδικας του επιτιθέμενου θα αποτύχει να εκτελεστεί στον τυχαιοποιημένο επεξεργαστή του απομακρυσμένου συστήματος. Παρόλαυτά, ενώ η τεχνική ISR είναι διαδεδομένη, όλες οι υπάρχουσες υλοποιήσεις της βασίζονται σε προσομοιωτές και άλλα εργαλεία τα οποία (i) επιφέρουν σημαντική μείωση στην απόδοση του επεξεργαστή, (ii) περιορίζουν την ευκολία ανάπτυξης τυχαιοποιημένου συνόλου εντολών, (iii) αδυνατούν να προστατέψουν τον πυρήνα του λειτουργικού συστήματος, και (iv) είναι επιρρεπή στις επιθέσεις που προσπαθούν να παρακάμψουν την τεχνική.

Για να αντιμετωπίσουμε αυτά τα προβλήματα προτείνουμε το ASIST: μια αρχιτεκτονική με υποστήριξη υλικού και λογισμικού-λειτουργικού συστήματος για την τυχαιοποίηση του συνόλου των εντολών του επεξεργαστή. Παραθέτουμε το σχεδιασμό και την υλοποίηση του ASIST τροποποιώντας έναν επεξεργαστή SPARC τον οποίο συνθέσαμε σε μία πλακέτα FPGA και τρέξαμε πάνω σε αυτό λειτουργικό σύστημα Linux το οποίο με τη σειρά του τροποποιήσαμε ώστε να υποστηρίζει τις καινούργιες μας λειτουργίες. Το λειτουργικό σύστημα φορτώνει ένα κλειδί, το οποίο χρησιμοποιείται για την τυχαιοποίηση των εντολών της τρέχουσας διαδικασίας σε έναν νέο καταχωρητή που έχουμε ορίσει, έπειτα ο τυχαιοποιημένος επεξεργαστής αποκωδικοποιεί τις εντολές της διαδικασίας με αυτό το κλειδί πριν την εκτέλεση τους. Επιπλέον, το ASIST προστατεύει το σύστημα ενάντια σε επιθέσεις που εκμεταλεύοντε αδυναμίες του πυρήνα του λειτουργικού συστήματος από το να εκτελέσουν αυθαίρετο κώδικα με αυξημένα προνόμια, χρησιμοποιώντας διαφορετικό κλειδί για το λειτουργικό σύστημα.

Στηρίζουμε πως το ASIST προστατεύει όλες τις εφαρμογές και τον πυρήνα του λειτουργικού συστήματος από επιθέσεις εμβόλιμου κώδικα με λιγότερο απο 1.5% μείωση στην απόδοση του τρέχοντος συστήματος, ενώ απαιτεί μόνο 0.7% παραπάνω κυκλώματα.

Επόπτης Καθηγητής: Ευάγγελος Μαρκατος

Acknowledgments

I am deeply grateful to my supervisor, Professor Evangelos Markatos, for his valuable advice and guidance during all my studies as well as Dr. Sotiris Ioannidis, Principal Reseascher, who has been the initiator and leading person who coordinated this work.

I am also grateful to Antonis Papadogiannakis for his constanct support, contribution and excellant cooperation. Many thanks to Vassilis Papaefstathiou for his valuable assistance, sharing his expertise on the deep hardware- level abyss. It is truly a unique experience to work with these people.

My best thanks to my friends and colleagues George Borbudakis, George Vassiliadis, Elias Athanasopoulos, Iasonas Polakis, Christos Papachristos, Michalis Polychronakis, Spiros Antonatos, past and current members of the Distributed Computing Systems Laboratory in ICS/FORTH and many others who I do not mention their name, for their inspiration, support and advice, and for sharing with me these years of my life.

Finally, I would like to thank my parents, for their support, patience and encouragement during all these years.

to my parents for their selfless love and support.

This work appeared on the proceedings of the 20th ACM conference on Computer & Communications Security (CCS13), November 2013 [40]

Contents

1 Introduction **1**

1.1 Contributions 5

1.2 Thesis Outline 6

2 Instruction Set Randomization **7**

2.1 Threat Model 7

2.1.1 Remote and Local Machine Code Injection Attacks. . . 7

2.1.2 Kernel vulnerabilities 8

2.1.3 Return-to-libc and ROP attacks. 9

2.1.4 Key Guessing Attacks. 9

2.2 Defense with ISR 9

2.3 Limitations of Existing Implementations 10

2.3.1 Existing Implementations 10

2.3.2 Limitations 11

3 ASIST Architecture **15**

3.1 Architecture Overview 15

3.2 Encryption 17

3.2.1 Static Binary Encryption 18

3.2.2 Dynamic Code Encryption 20

3.2.3 Shared Libraries 22

3.2.4 Self-modifying Code 22
3.2.5 Encryption Algorithms and Key Size 23
3.2.6 Tolerance to Key Guessing Attacks 23
3.3 Hardware Support 24
3.3.1 Placement of the Decryption Unit 26
3.3.2 Decryption Algorithms and Key Size 27
3.3.3 Return Address Encryption 28
3.4 Operating System Support 32
3.4.1 Kernel Modifications 32
3.4.2 Kernel Encryption 33

4 ASIST Prototype Implementation 35
4.1 Hardware Implementation 35
4.2 Additional Hardware 36
4.3 Kernel and Software Modifications 37
4.4 Portability to Other Systems 37

5 Experimental Evaluation 39
5.1 Security Evaluation 40
5.2 Performance Evaluation 42
5.2.1 Benchmarks 42
5.2.2 Real-world Applications 47
5.3 Memory Overhead 48
5.3.1 External Fragmentation 49

6 Related Work 51
6.1 Instruction Set Randomization 51
6.2 Other Defenses against Code Injection Attacks 52
6.3 Defenses against Buffer Overflow Attacks. 53
6.4 Other Randomization-based Defenses. 53
6.5 Hardware Support for Security 54

List of Figures

3.1 ASIST architecture 16
3.2 ELF format 18
3.3 Hardware module for runtime decryption 24
3.4 Decryption before and after the instruction cache 29
3.5 Decryption using XOR 128-bit key 31
3.6 Decryption using transposition 160-bit key 31

5.1 Runtime overhead on SPEC 43
5.2 lighttpd slowdown 46
5.3 sqlite3 slowdown 46

List of Tables

2.1 Comparison with existing ISR implementation 8

4.1 Addition hardware used by ASIST 36

5.1 Code injection attacks tested with ASIST 41

5.2 Data and text page faults per second in SPEC 44

5.3 sqlite3 slowdown 44

5.4 Text page faults in all 3 ASIST implementations 48

1 Introduction

Remote code injection attacks exploit software vulnerabilities to inject and execute arbitrary malicious code, allowing the attacker to obtain full access to the vulnerable system. There are several ways to achieve arbitrary code execution through the exploitation of a software vulnerability. The vast majority of code injection attacks exploit vulnerabilities that allow the diversion of normal control flow to the injected malicious code. Arbitrary code execution is also possible through the modification of non-control-data [18]. The most commonly exploited vulnerabilities for code injection attacks are buffer overflows [1]. Despite considerable research efforts [21,22,24,27,54], buffer overflow vulnerabilities remain a major security threat [19]. Other software vulnerabilities that allow the corruption of critical data are format-string errors [20] and integer overflows [26, 55].

Remotely exploitable vulnerabilities are continuously being discovered in popular network applications [9, 10] and operating system kernels [5, 6, 8, 17]. Thus, code injection attacks remain one of the most common security threats [4], exposing significant challenges to current security systems. For instance, the massive outbreak of the Conficker worm in 2009 infected more than 10 million machines worldwide [43]. Like most of the Internet worms, Conficker was based on a typical code injection attack that exploited a vulnerability in Windows RPC [7]. Along with the continuous discovery of new remotely exploitable vulnerabilities and zero-day attacks, the increasing complexity and sophisticated evasive methods of attack techniques [2, 25, 39] has significantly reduced the effectiveness of attack detection systems.

Instruction Set Randomization (ISR) [11, 12, 14, 31, 33, 44] has been proposed to defend against *any* type of code injection attack. ISR randomizes the instruction set of a processor so that an attacker is not able to know the processor's "language" to inject meaningful code. Therefore, any injected code will fail to accomplish the desirable malicious behavior, probably resulting in invalid instructions. To prevent successful machine code injections, ISR techniques encrypt the instructions of a program that may contain vulnerable software with a program-specific key. This key actually defines the valid instruction set for this specific 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 execution after decryption. Any injected code that is not encrypted with the correct key will result in irrelevant or invalid instructions.

Existing ISR implementations use binary transformation tools, such as `objcopy`, to encrypt the programs. For runtime decryption they use emulators [14, 33], like `Bochs` [35], and `Valgrind` [38], software dynamic translation tools [31], like `Strata` [45], or dynamic binary instrumentation tools [11, 12, 31, 44]. like `Valgrind` [38] and `PIN` [36]. However, the existing ISR implementations have several limitations: (i) They incur a significant runtime performance overhead due to the software emulator or instrumentation tool used for decryption. This overhead is prohibitive for a wide adoption of such techniques. (ii) Deployment is limited by the necessity of several tools, like emulators, and manual encryption of the programs that are protected with ISR. (iii) Existing implementations are vulnerable to code injection attacks into the underlying emulator or instrumentation tools. More importantly, they do not protect systems against attacks targeting remotely exploitable kernel vulnerabilities [5, 6, 8, 17], which are becoming an increasingly attractive target for attackers. Exploiting a kernel vulnerability may also allow for running user-level code with elevated kernel privileges [34]. (iv) Most ISR implementations are vulnerable to evasion attacks aiming to guess the encryption key and bypass ISR protection [51].

To address these issues we propose ASIST: a hardware/software scheme to support ISR on top of an unmodified ISA. Researchers have proposed hardware extensions to enhance security in the past [24, 30, 47, 54] including ISR [47], in the past. We advocate that hardware support for ISR is essential to guard against code injection attacks, at both user- and kernel-level, without incurring significant performance penalty at runtime.

ASIST uses distinct per-process keys and another key for the operating system kernel's code. To support runtime decryption at the CPU, we propose the use of two new registers in the ASIST-enabled processor: the *usrkey* and *oskey* registers, which contain the user- and kernel-level key of the running process. These registers are memory mapped and they are only accessible by the operating system via the privileged instructions *sta* and *lda* that store/load word to/from alternate space; our implementation for the SPARC architecture maps these registers into a new Address Space Identifier (ASI). The operating system is responsible for reading or generating the key of each program at load time, and associate it with the respective process. It is also responsible to store at the *usrkey* register the key of the next process scheduled for execution at a context switch. Whenever a trap to kernel is called, the CPU enters supervisor mode and the value of the *oskey* register is used to decrypt instructions. When the CPU is not in supervisor mode, it decrypts each instruction using the *usrkey* register.

We explore two possible choices for implementing the decryption unit at the instruction fetch pipeline of the modified processor. We also implement two different encryption algorithms, (i) XOR and (ii) Transposition, and use different key sizes. Additionally, we compare two alternative techniques for encrypting the executable code: (i) statically, by adding a new section in ELF that contains the key and encrypting all code sections with this key using a binary transformation tool, and (ii) dynamically, by generating a random key at load time and encrypting with this key all the memory mapped pages that contain code at the page fault handler. The dynamic encryption approach can support 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. Our modified processor can also encrypt the return address at each function call and decrypt it before returning to caller. In this way, ASIST protects the system from any stack-based attack targeting the return address.

To demonstrate the feasibility of our approach we present the prototype implementation of ASIST by modifying the Leon3 SPARC V8 processor [3], a 32-bit open-source synthesizable processor [28]. We mapped the modified processor to a Xilinx XUPV5 ML509 FPGA board [58]. 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

ASIST is able to prevent code injection attacks and buffer overflow exploits, which succeed on the vanilla system, practically without any performance overhead, while adding less than 1% of additional hardware to support ISR with our design. Our results also indicate that the proposed dynamic code encryption at the page fault handler does not impose any significant overhead, due to the low page fault rate for pages with executable code. This outcome makes our dynamic encryption approach very appealing, as it is able to *transparently* encrypt any executable program, it generates a different random key at each execution, and it supports shared libraries with negligible overhead.

1.1 Contributions

The main contributions of this work are:

- We propose ASIST: the first hardware-based support for ISR to prevent machine code injections without any performance overhead. We demonstrate the feasibility of hardware-based support for ISR by presenting the design, implementation, and experimental evaluation of ASIST.
- We introduce a dynamic code encryption technique that transparently encrypts pages with executable code at the page fault handler, using a randomly generated key for each execution. We show that this technique supports shared libraries and does not impose significant overhead to the system.
- We explore different choices for the decryption unit in hardware, we compare static and dynamic encryption, as well as different encryption algorithms and key sizes in order to improve the resistance of ISR against evasion attempts.

- We show that a hardware-based ISR implementation, like ASIST, is able to protect the system against attacks that exploit OS kernel vulnerabilities.
- We evaluated our prototype implementation with hardware-enabled ISR and show that it is able to prevent code injection attacks with negligible overhead.

1.2 Thesis Outline

The rest of the thesis is organized as follows: Chapter 2 explains our threat model, gives some background information about ISR, and discusses the limitations of current ISR implementations. In Chapter 3 we present the design of our proposed architecture: a modified processor with ISR support, a modified operating system to support the new hardware features, static and dynamic encryption of user-level applications, and encryption of kernel's code. We also discuss the different design choices for our system. Chapter 4 gives some details of our prototype implementation and synthesis of the new processor onto an FPGA. In Chapter 5 we experimentally evaluate the security and performance of the prototype implementation of our proposed architecture. Finally, Chapter 6 reviews prior work, and Chapter 7 concludes the thesis.

2 Instruction Set Randomization

In this section we describe our threat model, give some background on ISR, and discuss the main limitations of existing implementations that emphasize the need for hardware support.

2.1 Threat Model

2.1.1 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 code. We address vulnerabilities in the stack, heap, or BSS, e.g., any buffer overflow that overwrites the return address, a function pointer, or any control data. We focus on protecting

ISR Implementation	**Runtime Overhead**	**Shared Libraries**	**Self-modifying Code**	**Hardware Support**	**Encryption**	**Dynamic Encryption**	**Kernel Protection**	**ROP Prevention**
Bochs emulator [33]	High	No	No	No	XOR with 32-bit key	No	No	No
Valgrind tool [11, 12]	High	Yes	API	No	XOR with random key	Yes	No	No
Strata SDT [31]	Medium	No	No	No	AES with 128-bit key	No	No	No
EMUrand emulator [14]	Medium	No	No	No	XOR with 32-bit key	No	No	No
Pin tool [44]	Medium	Yes	Partially	No	XOR with 16-bit key	No	No	No
ASIST	Zero	Yes	API	Yes	XOR with 32-bit–128-bit key, Transposition with 160-bit key	Yes	Yes	Yes

Table 2.1: *Comparison of ASIST with existing ISR implementations.* ASIST provides a hardware-based implementation of ISR without runtime overhead, it supports the necessary features of current systems and protects against kernel vulnerabilities.

the potentially vulnerable systems against *any* type of machine code injection attacks. In a typical binary code injection attack, the attacker sends a malicious input that exploits a memory corruption vulnerability on the victim's computer, which permits remote execution of arbitrary code and even complete takeover of the system. This code is usually supplied by the attacker as a part of malicious input, and the control flow is driven at this input.

2.1.2 Kernel vulnerabilities

Remotely exploitable vulnerabilities on the operating system kernel [5, 6, 8, 17] are becoming an increasingly attractive target for attackers. Our threat model includes code injection attacks based on kernel vulnerabilities. We propose an architecture that is capable of protecting the operating system kernel as well. We also address attacks that use a kernel vulnerability to run user-level codei, *return-to-user attacks*, with elevated kernel privileges [34].

2.1.3 Return-to-libc and ROP 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: re-direct the execution to existing library functions, attacks typically known as *return-to-libc* attacks [37], or use existing instruction sequences ending with a *ret* instruction (called *gadgets*) to implement the attack, a technique known as *return-oriented programming* (ROP) [15,48]. Although ISR protects a system against any type of code injection attacks, its threat model does not address return-to-libc and ROP attacks that use existing code to harm or takeover a system. Existing implementations of ISR follow this threat model so they do not aim to protect the system from return-to-libc attacks. However, due to the rise of such attacks, we aim to protect systems from return-to-libc attacks using the same hardware.

2.1.4 Key Guessing Attacks.

Existing ISR implementations are vulnerable to key guessing or key stealing attacks [51, 57]. 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.

2.2 Defense with ISR

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 [11]. 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. Any injected instruction sequences 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.

2.3 Limitations of Existing Implementations

2.3.1 Existing Implementations

Existing ISR Implementations [11, 12, 14, 31, 33, 44] use binary transformation tools, such as `objcopy`, to encrypt the code of user-level programs that will be protected. For runtime decryption they use emulators [35] or dynamic binary instrumentation tools [36, 38, 45]. In Table 2.1 we list and compare all the existing ISR implementations.

Kc et al. [33] implemented ISR by modifying the `Bochs` emulator [35] to decrypt at runtime the code of statically encrypted programs, 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 while this system does not support shared libraries and self-modifying code. Barrantes et al. [11, 12] use `Valgrind` [38] to decrypt applications' code, which is encrypted with XOR and a random key equal to the program's length when applications are loaded into Valgrind. 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. [31] implemented ISR with a software dynamic translation tool [45] using AES encryption with 128-bit key size to prevent attacks trying to guess the encryption key [51]. 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. [14] 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 [44] implemented ISR with shared libraries support using Pin binary instrumentation tool [36]. 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.

2.3.2 Limitations

The main limitations of the existing ISR implementations are:

1. *High runtime performance overhead.* All the existing implementations of ISR have a considerable runtime overhead, which becomes significantly higher for CPU-intensive applications. This is because all the proposed systems use extra software to emulate or translate the instructions before they are executed, which results to more instructions and increased execution times. We argue that the most efficient approach is a hardware-based implementation of ISR.
2. *Deployment difficulties.* The need for several tools, such as emulators and binary instrumentation tools, as well as the need for manual encryption of all programs that will be protected, and the partial support for shared libraries limit the ease of deployment of ISR. On the other hand, we aim to build a system that will be able to transparently protect any program running on it without any modifications.
3. *Cannot protect kernel vulnerabilities.* None of the existing ISR prototype implementations is able to defend against attacks exploiting kernel vulnerabilities [5,6,8,17,34]. Such attacks are getting increasingly popular and allow attackers to run code with kernel privileges so we would like to protect the operating system kernel as well. Moreover,

the underlying emulators or instrumentation tools may be vulnerable to buffer overflow and code injection attacks. Although Pin has been extended with PinOS to instrument kernel's code as well [16], it has not been used to implement ISR support for the kernel. Even in this case, the code of PinOS would not be protected, while the use of a virtual machine in PinOS would impose a significant performance overhead.

4. *Cannot prevent ROP attacks.* ISR cannot protect a vulnerable randomized program against ROP attacks [15, 48], which use existing code to harm the system. This is because ISR was proposed to prevent code injection attacks, not code-reuse attacks that using existing code. Thus, the existing implementations follow the same threat model. However, due to the rise of such attacks recently, we would like to be able to easily extend an ISR system in order to provide defenses against ROP attacks as well.
5. *Evasion attacks by guessing the encryption key.* Many of the proposed ISR implementations are vulnerable to evasion attacks that try to guess the encryption key and inject valid code into the vulnerable system [51, 57]. as they use a constant key for each program among multiple executions. Sovarel et al. [51] demonstrate the feasibility of an incremental attack that uses partial key guessing to reduce the number of tries needed to find the key. Also, attackers may be able to steal or infer the encryption key when memory secrecy breaks [53, 57].

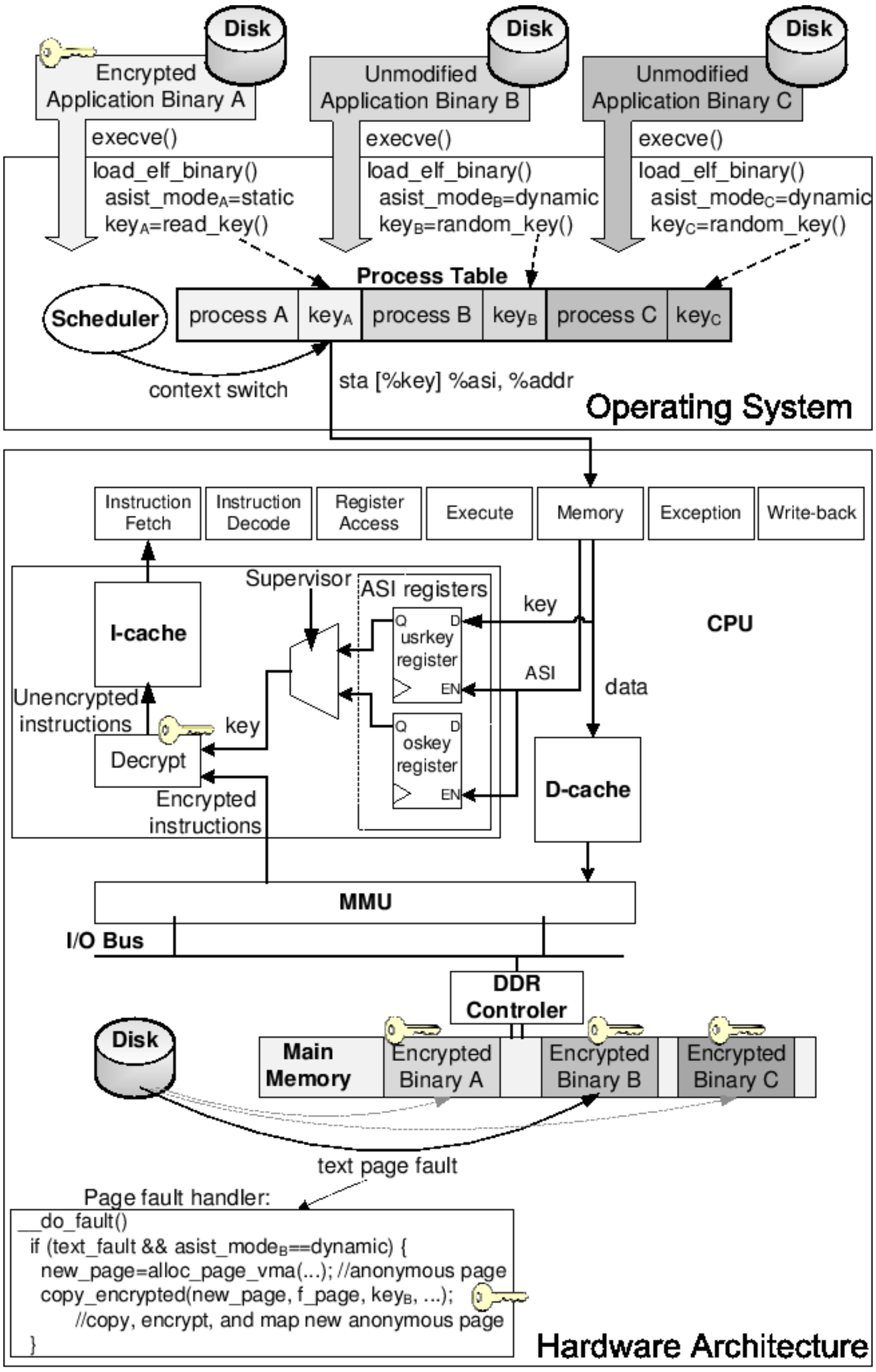
To put our work into context, we compare ASIST with other ISR implementations in Table 2.1. ASIST is the only ISR implementation with hardware support, resulting in negligible runtime overhead for any type of applications. ASIST also supports a new dynamic code encryption approach that allows the transparent encryption of any application with shared libraries. To defend against attempts to guess or steal the encryption key, ASIST (i) stores the encryption key in a hardware register accessible only by the kernel through privileged instructions, and avoids storing the key in process's memory, (ii) generates a new random key at each execution of the same program when dynamic encryption is used, (iii) supports large key sizes up to 128-bit, and (iv) besides XOR and transposition (already implemented), it supports more secure encryption algorithms in case of memory disclosure. Moreover, ASIST prevents the execution of injected code at the kernel, e.g.due to a kernel vulnerability, by using separate keys for user-level programs and kernel's code. Finally, ASIST is able to prevent ROP attacks targeting the return address by encrypting the return address in the modified processor.

3 ASIST Architecture

We now describe the design of the ASIST architecture in hardware and operating system, as well as the approach we use on code encryption.

3.1 Architecture Overview

Our architecture spans hardware, operating system and user space (see Figure 3.1), to support hardware-assisted ISR. ASIST supports two alternative ways of code encryption: static and dynamic. In static encryption, the key is pre-defined and exists within the executable file, while all code sections are already encrypted with this key. In case of dynamic encryption, the executable file is unmodified and the key is decided randomly by the respective loader of the operating system at the beginning of each execution. The code sections are encrypted dynamically at runtime whenever a code



```
execve()
load_elf_binary()
  asist_modeA=static
  keyA=read_key()
```

```
execve()
load_elf_binary()
  asist_modeB=dynamic
  keyB=random_key()
```

```
execve()
load_elf_binary()
  asist_modeC=dynamic
  keyC=random_key()
```

```
__do_fault()
  if (text_fault && asist_modeB==dynamic) {
    new_page=alloc_page_vma(...); //anonymous page
    copy_encrypted(new_page, f_page, keyB, ...);
        //copy, encrypt, and map new anonymous page
  }
```

Figure 3.1: *ASIST architecture.* 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. using the *sta* instruction. The processor decrypts each instruction with *usrkey* or *oskey* register, according to the *supervisor* bit. The decryption fits into the pipeline before instructions are fetched and stored in the instruction cache, without spending an extra cycle.

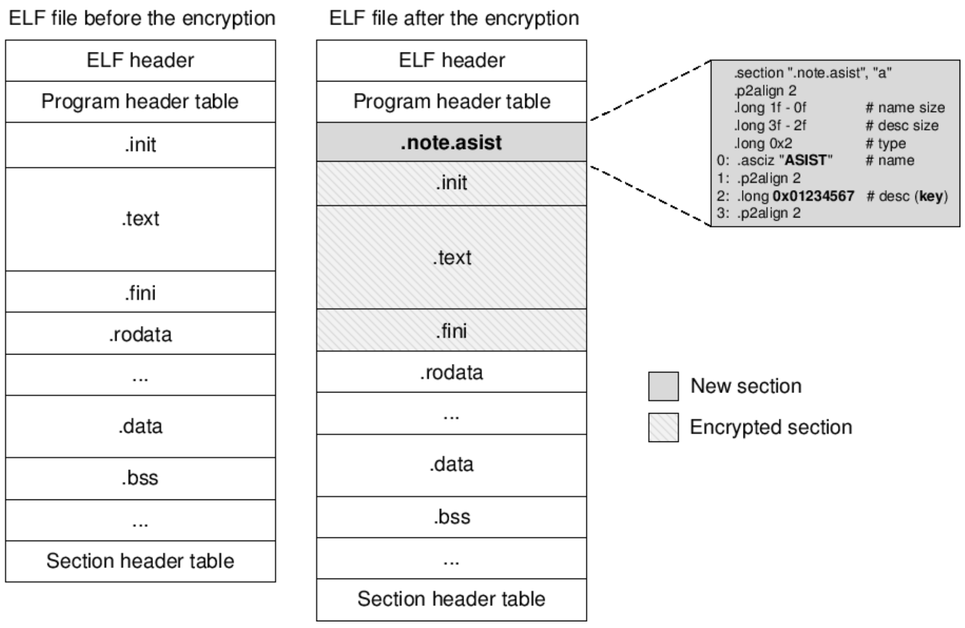
page is requested from file system and before this page gets mapped to the process's virtual address space.

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. The operating system keeps the key of each process in a respective field in the process table, and stores the key of the next process that is scheduled for execution in the *usrkey* register using the *sta* privileged SPARC instruction. Moreover, the processor is modified to decrypt each instruction before the instruction fetch cycle, using one of the above two registers as a key, according to the supervisor bit. In the following of this section we describe this design in more detail.

3.2 Encryption

We first describe the approaches we follow for the encryption of an executable program. We support two possible options for encrypting an executable program: static and dynamic encryption. In static encryption, the program is encrypted before each execution with a pre-defined key. using a modified `objcopy` binary transformation tool. 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 code encryption. 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, it needs to manually run the encryption program once per each executable file 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



```
    .section ".note.asist", "a"
    .p2align 2
    .long 1f - 0f          # name size
    .long 3f - 2f          # desc size
    .long 0x2              # type
0:  .asciz "ASIST"         # name
1:  .p2align 2
2:  .long 0x01234567       # desc (key)
3:  .p2align 2
```

Figure 3.2: *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.

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 a potential runtime overhead to encrypt a code page when it is loaded from the file system to memory at a code page fault. In Chapter 5 we experimentally evaluate this overhead and show that due to the low number of code page faults, dynamic encryption is very efficient.

3.2.1 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, else it is randomly chosen by the tool. Figure 3.2 shows the modifications of a statically encrypted ELF executable file. 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 i.e., the process that called the `execve()` system call to load this binary file. The existence of the `.note.asist` section indicates that the file has been statically encrypted with the given key. In this mode of operation 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 update the *usrkey* hardware register each time this process is scheduled for execution, so that the modified processor will properly decrypt its instructions.

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 during the creation of the original ELF 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, with static encryption, which may facilitate a key guessing attack, one approach could be to re-encrypt the binary after a process crash. Another 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 using the same approach we described above, meaning adding a note section with the new key and encrypting the code sections with this key in the new binary. This approach will also work only with statically linked libraries. However, we do not recommend this approach due to the extra time needed to copy and encrypt the entire binary at load time, especially for large binaries that are also statically linked with large libraries. Encrypting the entire binary is probably an unnecessary overhead, as many parts of the code that will be encrypted are unlikely to be actually executed. Note that for this reason the binary loaders do not read the entire binary from disk to load into memory; they usually memory map the binary file to the process's virtual address space and read code pages from disk as needed.

3.2.2 Dynamic Code Encryption

We now introduce a new technique to dynamically encrypt a program's code before it is loaded into the process's memory. This approach 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 the first time it is accessed by the program through a page fault. Thus, we decided to perform the code encryption at this point. This way, ASIST encrypts only the code pages that are actually used by the program at each execution.

First, the ELF binary loader is modified to randomly generate a new key, which is stored into the process table as a record of the current process. It also sets the `asist_mode` field of the current process to dynamic in order to distinguish processes running a statically encrypted binary from processes using dynamic encryption. 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. so the encrypted code is found at the address requested by the program.

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 due to the code encryption will not be stored at the original binary file. This technique, encryption aside, is similar to copy-on-write technique. Although processes running 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 different keys and so different representations of the same code in memory. 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. This memory overhead could be avoided if all processes running the same code use the same key, which enables an effective sharing of encrypted code pages, in expense though of weaker security against key guessing and other attacks that may compromise the key. In our implementation, we choose to use a different key per process and a small memory overhead.

In practice, the memory allocated for code accounts only for a small fraction of the total memory used by the process. Note that we can still benefit from buffer cache, to avoid unnecessary disk accesses, as we copy the cached page.

In order to use a different key per each process that use the dynamic encryption mode, 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. Thus, a page fault will occur at the first code access and the code pages will be encrypted with the child'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. In this way we can easily separate code from data and headers at different pages.

3.2.3 Shared Libraries

Our dynamic code encryption technique supports the use of shared libraries without extra effort. The code of a shared library 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.

3.2.4 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 (or change existing code). A randomized program that generates unencrypted code and then tries to execute it will result in invalid instructions executing, as the unencrypted code will be decrypted by the ASIST processor with the program's key. Thus, the unencrypted code cannot be executed. To support such programs, we added a new system call in Linux kernel: `asist_encrypt(char *buf, int size)`. This system call encrypts the code that exists in the memory region starting from `buf` with `size` bytes length, using the current process's key that is stored in process table as a record of the current process. However, the `buf` buffer may 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 [11], we believe that programs should carefully use the `asist_encrypt()` system call to avoid malicious code injection in `buf`.

3.2.5 Encryption Algorithms and Key Size

We now discuss the encryption algorithms and key size we can use in ASIST. 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. To accelerate encryption we XOR code and key as 32-bit words, instead of bits. The same algorithm is used for decryption with the same key. However, XOR was found to be susceptible to key guessing and key extraction attacks [51, 57]. In our prototype we implemented XOR encryption with key size that can range from 32-bit to 128-bit, to reduce the probability of a successful guess. The key size should be a multiple of 32-bit i.e., 32-bit, 64-bit, 96-bit, or 128-bit, to support XOR between 32-bit words.

We also implemented *transposition*, which is a stronger encryption algorithm than XOR. In transposition we shuffle the bits of a 32-bit word using an 160-bit key (also called as transposition table). 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 which are used to choose one of 32 bits at each position. The same key is used for decryption. We use the `asist_mode` flag to define the encryption algorithm, key size, and encryption method (static or dynamic). This flag exists in the process table records, and may also exist in the `.note.asist` ELF section of a statically encrypted binary.

3.2.6 Tolerance to Key Guessing Attacks

To evade ISR protection, an attacker can try to guess the encryption key and inject code encrypted with this key. The probability of a successful guess with XOR encryption is $1/2^{key\ size}$, e.g., $1/2^{32}$ for 32-bit key and $1/2^{128}$ for 128-bit key. In case of transposition, the probability of a successful guess is $1/32!$, which is much lower than the respective probability with XOR.

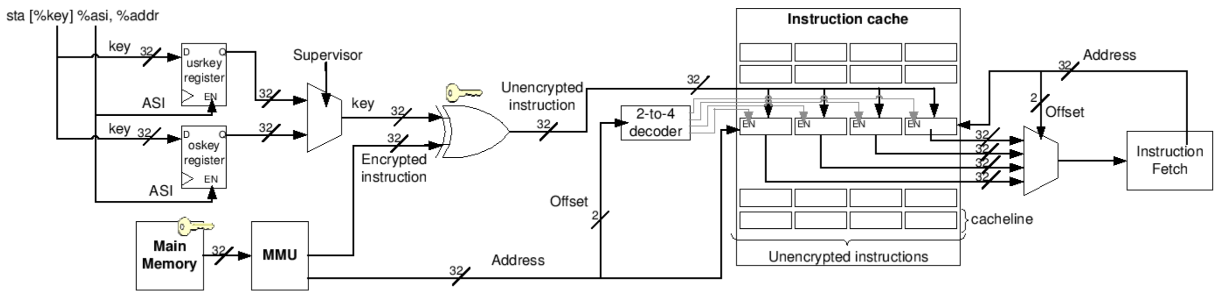


Figure 3.3: *ASIST hardware support for runtime instruction decryption.* We see the modified ASIST processor that decrypts every instruction with XOR and 32-bit key before the instruction cache. The key of the user-level running process is stored in *usrkey* register, and operating system's key is stored in *oskey* register using the *sta* privileged SPARC instruction. The supervisor bit defines which of these two keys will be used.

In case of a single guess, all the above probabilities seem good enough to protect a system. However, if the same key is used consistently, e.g., in case of static encryption, without re-encryption, a brute force attack can be used to eventually guess the correct key. Sovarel et al. [51] present an incremental attack that reduces the number of tries needed to find the encryption key by observing system's behavior. ASIST can address such attacks with dynamic encryption, as a new key is generated before each execution. Barrantes et al. [11] show that code injections in systems protected with ISR result in the execution of at most five instructions before causing an exception. Therefore, as dynamic encryption changes the key at each program execution, a brute force or incremental attack cannot succeed and the probability of success of a brute force or incremental attack remains $1/2^{key\ size}$ with XOR or $1/32!$ with transposition.

3.3 Hardware Support

We now discuss in more detail the hardware support provided by ASIST for runtime instruction decryption. Figure 3.3 outlines ASIST's hardware architecture for ISR support when using XOR with a 32-bit key. We added two new registers to store the encryption keys: *usrkey* for the key of the user-level running process, and *oskey* for the operating system's key. 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). We reserved a free ASI for these registers, that should be given in these instructions. The value of the key that will be stored with *sta* is first placed into a register, which is given in this instruction. Another register given to *sta* contains the address, that is 0 for *usrkey* and 32 for *oskey*.

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 than one *sta* instructions may be needed.

The ASIST 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*. When *rett* is called, and control returns back to user-level, the proper key of the running process is already at the *usrkey* register.

The decryption unit is placed before the instruction fetch cycle, when instructions are moved from memory to the instruction cache. We should note that decryption fits in the processor's pipeline and *no* extra cycle is spent on it. Therefore, we expect no runtime overhead from the hardware decryption part. We expect a slight increase on the cost and on the power consumption due to the extra hardware we used. Also, ASIST's hardware architecture is backwards compatible with programs and operating system kernels that are not encrypted. We set the default value of the key registers to zero, which has no effect on the decryption (the default decryption algorithm is XOR). Any unencrypted program with an unmodified kernel will be normally executed on the ASIST processor, but without ISR protection. In the rest of this section we discuss alternative choices for the placement of the decryption unit, and support for different decryption algorithms and key sizes.

3.3.1 Placement of the Decryption Unit

We decided to place the decryption unit as early as possible in the modified processor to avoid adding any performance overhead or spend an extra cycle, and to avoid breaking any runtime optimizations, like branch prediction, made by the processor. There are two possible choices for placing the decryption unit: before and after the instruction cache. Figure 3.4 presents the two options. We implemented both cases and both use the same amount of extra hardware, while none of them add any runtime overhead. When the decryption unit is after the instruction cache, the instructions are stored encrypted in cache and the decryption takes place at each fetch cycle. Therefore, it is on the critical path of the processor and although it may not add any observable delay for simple decryption schemes, it may add a delay for more complex decryption algorithms. Also, as the decryption circuit is utilized at each fetch cycle, it may result in increased power consumption. However, this approach protects the system from a possible code injection in the instruction cache. as the instructions remain encrypted in the cache and they are always decrypted in the path from the instruction cache to the fetch and execution.

On the other hand, when the decryption unit is located before the instruction cache, it is accessed only on instruction cache misses. Thus, the decryption circuit is used significantly fewer times than in the previous case. This leads to reduced power consumption for decryption, as the instructions that are executed many times, e.g., in loops, are found decrypted in the instruction cache. Also, an increased delay for more complex encryption at this point will not have significant impact to the overall performance of the processor as in the previous case. In this case, instructions are stored unencrypted into the instruction cache, which could be vulnerable to code injections in the instruction cache. However, to the best of our knowledge, it is not possible to inject code in the instruction cache without passing from the path we have modified to decrypt each instruction. For this reason, we selected to place the decryption unit before the instruction cache.

3.3.2 Decryption Algorithms and Key Size

As we explained in Section 3.2.5, we implemented two different encryption algorithms: XOR and transposition and consequently two respective decryption algorithms in hardware. The XOR decryption with a 32-bit key is simple to implement, as we illustrate in Figure 3.3. We also support different key sizes for XOR, from 32 to 128 bits.

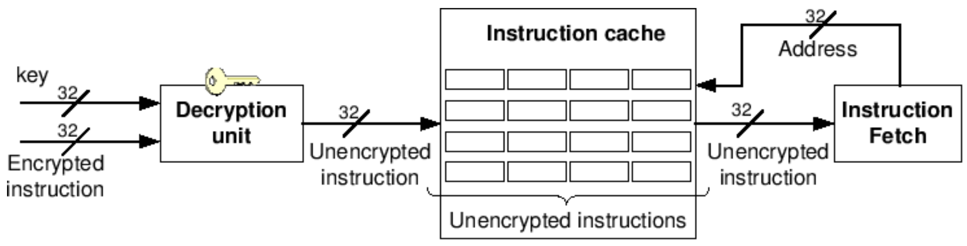
Figure 3.5 shows the implementation of XOR decryption with 128-bit key. Since each encrypted instruction in our architecture (SPARC V8) is a 32-bit word, we need to select the proper 32-bit part of the 128-bit key, the same part that was used in the encryption of this instruction. Thus, we use the two last bits of the instruction's address (word offset) 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 3.6, requires significantly 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 that corresponds to the bit of the original (decrypted) instruction. It also has 5 select lines that define the selection of the input bit at each position. The 5 select lines of each multiplexer are a 5-bit part of the 160-bit key. that is used with transposition. Besides the additional hardware, the runtime operation of transposition is equally fast with XOR, as it does not spend an extra cycle and does not impose any delay to the processor.

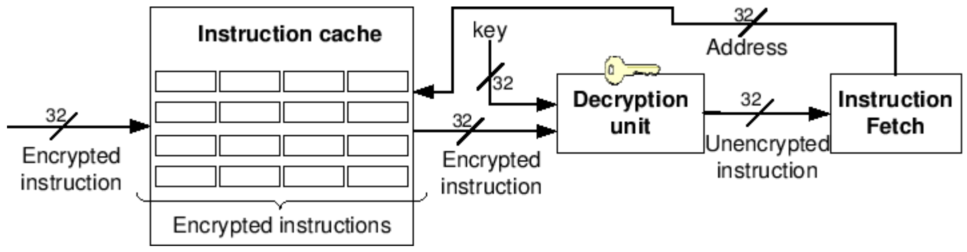
To dynamically select the decryption algorithm and key size, we have added another memory mapped register: *asist_mode*. This register can be set by the operating system to define the decryption algorithm and key size that will be used in the hardware.

3.3.3 Return Address Encryption

The design we have presented so far is able to efficiently protect the system against any type of binary code injection attacks, using ISR. To transparently protect a system against return-to-libc and ROP attacks [15, 48], we extended our hardware design to provide protection of the return address integrity without any runtime overhead. To this end, we slightly modified the ASIST processor to encrypt the return address in each function call using the process's key, and decrypt it just before returning to the caller. This is similar to the XOR random canary defense [22], which uses `mprotect()` to hide the canary table from attackers. On the other hand, we take advantage of the two hardware key registers, which are not accessible by an



(a) Decryption before the instruction cache



(b) Decryption after the instruction cache

Figure 3.4: *Alternative choices for the placement of the decryption unit in the ASIST-enabled processor.*

attacker, to hide the encryption key. Also, our hardware implementation does not impose any performance overhead.

In the SPARC V8 architecture, function calls are performed with the *call* synthetic instruction, which is equal to *jmpl func_addr,%o7*. Hence, *call* writes the contents of the program counter (PC), i.e., the return address, into the *o7* register, and then transfers the control to the function's address *func_addr*. To return from a function, the *ret* synthetic instruction is used, which is equal to *jmpl %i7+8,%g0* when returning from a normal subroutine (*i7* register in the callee is the same with *o7* register in the caller) and *jmpl %o7+8,%g0* when returning from a leaf subroutine.

To encrypt the return address on each function call in the modified processor, we just XOR the value of the PC with the *usrkey* register when a *call* or *jmpl* instruction is executed and the value of the PC is stored into the *o7* register. The return address, i.e., the *i7* register in the callee, is decrypted with *usrkey* when a *jmpl* instruction uses the *i7* register (or *o7* in case of leaf subroutine) to change the control flow (*ret* instrunction). Thus, the modified processor will return to the *(%i7 XOR usrkey)+8* address.

This way, the return address remains always encrypted, e.g., when it is pushed onto the stack (window overflow), and it is always decrypted by the *jmpl* instruction when returning. Hence, any modification of the return address, e.g., though a stack-based buffer overflow or fake stack by changing the stack base pointer, or any *ret* instructions executed by a ROP exploit without the proper *call*, will lead to an unpredictable return address upon decryption, as the *usrkey* is unknown to the attacker.

Note that *jmpl* is also used for indirect jumps, not only for function calls and return, so our modified *jmpl* decrypts the given address only when the *i7* (or *o7*) register is used. This is a usual convention for function calls in SPARC and it should be obeyed, i.e., the *i7* and *o7* registers should not be used for any indirect jumps besides returning from function calls. Also, the calling conventions should be strictly obeyed: return address cannot be changed in any legal way before returning, and *ret* instructions without a preceding *call* instruction cannot be called without a system crash. As the calling conventions are not always strictly obeyed in several legacy applications and libraries, the use of return address encryption may not be always possible. Therefore, although ASIST offers this hardware feature, it may or may not be enabled by the software. We use one bit of the *asist_mode* register to define whether the return address encryption will be enabled or not.

A similar approach is implemented by Tuck et al. [54] to protect function pointers from buffer overflows, and it could be integrated with ASIST to thwart more types of return-to-libc attacks, besides attacks targeting the return address.

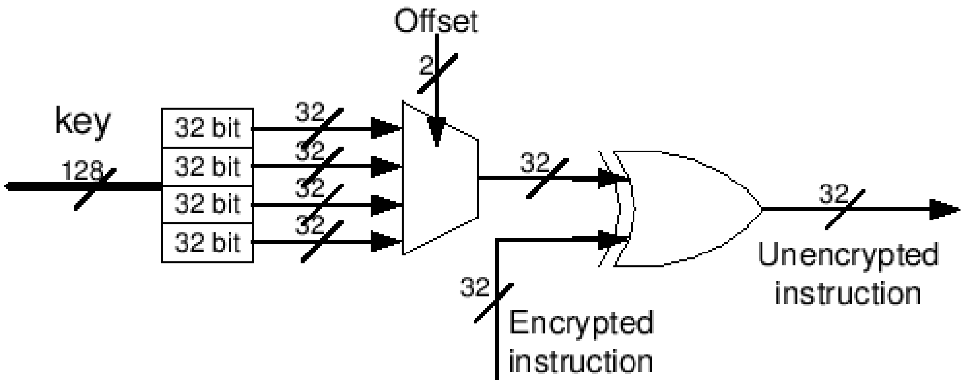


Figure 3.5: *Decryption using XOR with 128-bit key.* Based on the last two bits of the instruction's address (offset) we select the respective 32-bit part of the 128-bit key for decryption.

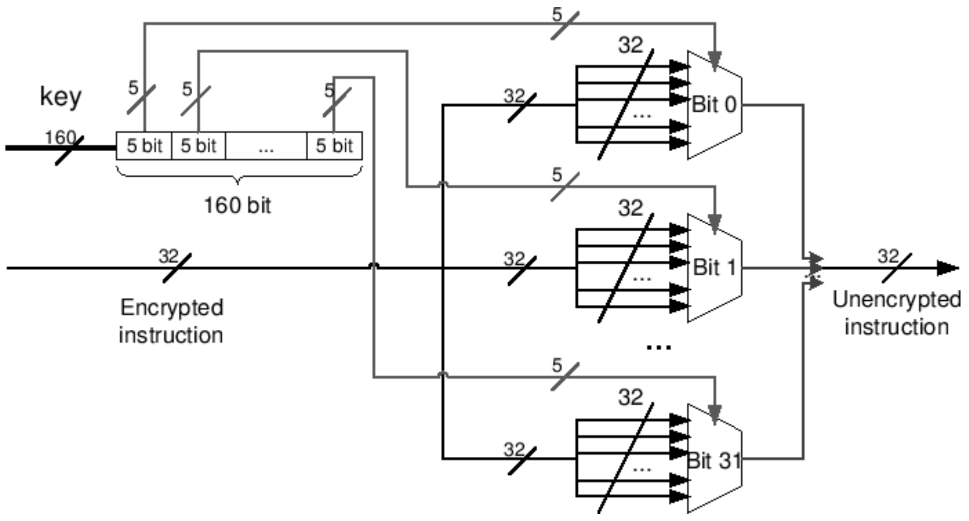


Figure 3.6: *Decryption using transposition with 160-bit key.* The implementation of transposition requires significantly more hardware, because it needs 32 multiplexers (one per each bit of the decrypted instruction) with all the 32 bits of the encrypted instruction as input lines in each one. Each multiplexer uses a 5-bit part of the key as select line.

3.4 Operating System Support

We now describe the new functionality we added in the operating system to support the ASIST hardware features for ISR in order to protect the system from attacks against possibly vulnerable user-level processes and kernel's vulnerabilities.

3.4.1 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 first 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 and map 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. Eventually, this page will contain the code that will be accessed by the process.

3.4.2 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 carefully separate code from data into different sections while building the kernel image. The *oskey* register saves the key of kernel's encrypted code. We modified the bootloader to read and then store the kernel's key into the *oskey* register with an *sta* instruction, just before the control is transfered from bootloader to kernel. 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.

We decided to statically encrypt the kernel's code so as to not add any delay to the boot process. Due to this, the key is decided once when 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 at this point. This option could add a further delay to the boot process. However, 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 and then encrypt the kernel's code along with decompression.

4

ASIST Prototype Implementation

In this section we describe the ASIST prototype implementation and hardware synthesis using an FPGA board, We also present the results of the hardware synthesis in terms of additional hardware needed for our prototype, comparing with the unmodified processor. Finally, we discuss how the proposed system that we have implemented by modifying the SPARC V8 Leon3 processor can be easily ported to other hardware architectures, such as x86, and other operating systems.

4.1 Hardware Implementation

We modified Leon3 SPARC V8 processor [3], a 32-bit open-source synthesizable processor [28], to implement the security features of ASIST for hardware-based ISR support, as we described in Section 3.3. All hardware

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)

Table 4.1: *Additional hardware used by ASIST.* We see that ASIST 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.

modifications for instruction decryption, required fewer than 100 lines of VHDL code.

Leon3 uses a single-issue, 7-stage pipeline. Our implementation has 8 register windows, an 16 KB 2-way set associative instruction cache, and a 16 KB 4-way set associative data cache. We synthesized and mapped the modified ASIST processors on a Xilinx XUPV5 ML509 FPGA board [58]. The FPGA has 256 MB DDR2 SDRAM memory and the design operates at 80 MHz clock frequency. It also has several peripherals including an 100Mb Ethernet interface.

4.2 Additional Hardware

Table 4.1 shows the results of the synthesis for three different hardware implementations of ASIST, using XOR decryption with 32-bit and 128-bit keys, and decryption with transposition using 160-bit key. We compare them with the unmodified Leon3 processor as a baseline to measure the additional hardware used by ASIST to implement ISR functionality in each case. We see that ASIST 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, with 32 multiplexer with 32 input lines each, as we showed in Section 3.3.2.

4.3 Kernel and Software Modifications

The resulting system is a full-featured SPARC workstation using a Linux operating system. We modified the Linux kernel as we described in Section 3.4. We ported our Linux kernel modifications in 2.6.21 and 3.8.0 kernel versions. Although a single kernel is able to support both static and dynamic encryption, depending on the existence of the ASIST note section in the ELF, in order to separately evaluate the two approaches we had three different versions of each kernel: an unmodified one, a kernel with static encryption support only, and a kernel version only with dynamic encryption support.

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 ASIST), including vulnerable programs and benchmarks, were cross compiled with this tool chain in another PC. We slightly modified linker scripts 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. To implement static encryption, we extended `objcopy` with the `---encrypt-code` flag to add the new note section and encrypt code sections. The key can be provided by the user or be randomly chosen by the tool.

4.4 Portability to Other Systems

Our approach is easily portable to other architectures and operating systems. Regarding ASIST's hardware extensions, implementing new registers that are accessible by the operating system is quite easy 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. In our current design, we have implemented the runtime instruction decryption for RISC architectures that use fixed-length instructions. Thus, porting the decryption functionality in other RISC systems will be straight-forward. On the other hand, CISC architectures such as x86 support variable-length instructions. However, our approach can also be implemented in such architectures with minor modifications. Since instructions reside in memory before they are executed, we can simply encrypt them without the need of precise disassembly, e.g., in blocks of 32-bits, depending on the key size. In architectures with variable-length instructions this encryption will not be aligned at each instruction, but this is not an issue. The memory blocks will be decrypted accordingly by the modified processor before execution. For instance, a memory block can be decrypted based on the byte offset of its respective memory address. Also, since we have placed the decryption unit before the instruction cache, decryption is performed at each *word* that is stored in cache, rather than at each instruction.

We have implemented our prototype by modifying the Linux kernel. However, the same modifications can be made in other operating systems as well, as we change generic kernel modules such as the binary loader, the process scheduler and context switch, and the page fault handler. These modules exist in all modern operating systems and they can be changed respectively to support the hardware features offered by a randomized processor.

5 Experimental Evaluation

We mapped our prototype onto an FPGA running two versions of the Linux kernel, 2.6.21 and 3.8, as described in Chapter 4. We used the Ethernet adapter of the FPGA and configured the system with networking and a static IP address. This allows for remote exploitation attempts for our security evaluation, and for evaluating the performance of a Web server. As the available memory on the FPGA is only 256 MB, and there is no local disk in the system, we used NFS to mount a partition of a local PC that contains all the cross compiled programs needed for the evaluation. To avoid measuring NFS delays in our evaluation, we copied each executable program in the local RAM file system before its execution.

We evaluated the ASIST prototype that uses XOR encryption with a 32-bit key, comparing static and dynamic encryption implementations with an unmodified system (vanilla processor and unmodified operating system). We observed that using a larger key or transposition instead of XOR for encrypting instructions has the same effectiveness on preventing code injection attacks and the same efficiency in terms of performance. We did not use the return address encryption in our security and performance evaluation.

5.1 Security Evaluation

To demonstrate the effectiveness of ASIST at preventing code injection attacks exploiting user- or kernel-level vulnerabilities, we tested a representative sample of attacks shown in Table 5.1. The first six attacks target buffer overflow vulnerabilities on user-level programs, while the last three attacks target a NULL pointer dereference and two buffer overflow vulnerabilities of the Linux kernel.

First, we ran a vanilla 2.6.21 kernel, which does not properly implement a non-executable stack on SPARC. We built a custom program with a typical stack-based buffer overflow vulnerability, and we used a large command-line argument to inject SPARC executable code into the program's stack, which was successfully executed by overwriting the return address. We then used an ASIST modified kernel without enabling the return address encryption, and we ran a statically encrypted version of the vulnerable program with the same argument. In this case, the program was terminated with an illegal instruction exception, as the unencrypted injected code could not be executed. Similarly, we ran 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.

CVE Reference	Vulnerability Description	Access Vector	Location	Vulnerable Program
CVE-2010-1451	Linux kernel before 2.6.33 does not properly implement a non-executable stack on SPARC platform	Local	Stack	Custom
CVE-2013-0722	Buffer overflow due to incorrect user-supplied input validation	Remote	Stack	Ettercap 0.7.5.1 and earlier
CVE-2012-5611	Buffer overflow that allows remote authenticated users to execute arbitrary code via a long argument to the GRANT FILE command	Remote	Stack	Oracle MySQL 5.1.65 and MariaDB 5.3.10
CVE-2002-1549	Buffer overflow that allows to execute arbitrary code via a long HTTP GET request	Remote	Stack	Light HTTPd (lhttpd) 0.1
CVE-2002-1337	Buffer overflow that allows to execute arbitrary code via certain formatted address fields	Remote	BSS	Sendmail 5.79 to 8.12.7
CVE-2002-1496	Buffer overflow that allows to execute arbitrary code via a negative value in the Content-Length HTTP header	Remote	Heap	Null HTTPd Server 0.5.0 and earlier
CVE-2010-4258	Linux kernel allows to bypass access_ok() and overwrite arbitrary kernel memory locations by NULL pointer dereference to gain privileges	Local	Kernel	Linux kernel before 2.6.36.2
CVE-2009-3234	Buffer overflow that allows to execute arbitrary user-level code via a ”big size data“ to the perf_counter_open() system call	Local	Kernel stack	Linux kernel 2.6.31-rc1
CVE-2005-2490	Buffer overflow that allows to execute arbitrary code by calling sendmsg() and modifying the message contents in another thread	Local	Stack	Linux kernel before 2.6.13.1

Table 5.1: *Representative subset of code injection attacks tested with ASIST.* We see that ASIST is able to successfully prevent code injection attacks targeting vulnerable user-level programs as well as kernel vulnerabilities.

We performed similar tests with all the other vulnerable programs: Ettercap, which is a packet capture tool, MariaDB database, Light HTTPd and Null HTTPd webservers, and sendmail. These programs were cross compiled with our toolchain and encrypted with our extended `objcopy` tool. In all cases our remotely injected shellcode was executed successfully only on the vanilla system, while ASIST always prevented the execution of the injected code and resulted in illegal instruction exception.

We also tested attacks exploiting three kernel vulnerabilities with and without ASIST. We cross compiled, modified and encrypted three different kernel versions for each one: 2.6.21, 2.6.31-rc1 and 2.6.11. When running the vanilla kernel on the unmodified processor, the kernel exploits resulted in the successful execution of the provided user-level code with kernel privileges. On the other hand, the encrypted kernels with ASIST resulted in kernel panic for all the exploits, avoiding a system compromise with kernel privileges.

5.2 Performance Evaluation

To evaluate ASIST's performance we compare (i) vanilla Leon3 with unmodified Linux kernel (Vanilla), (ii) ASIST with static encryption (ASIST-Static), and (iii) ASIST with dynamic code encryption (ASIST-Dynamic), when running the SPEC CPU2006 benchmark suite and two real world applications.

5.2.1 Benchmarks

We ran all the integer benchmarks from the SPEC CPU2006 suite (CINT2006) [52], which includes several CPU-intensive applications. Figure 5.1 shows the slowdown of each benchmark when using ASIST with static and dynamic encryption, compared to the vanilla system. We see that both ASIST implementations impose less than 1.5% slowdown in all benchmarks. For most benchmarks, ASIST exhibits almost the same execution times as with the unmodified system. This is due to the hardware-based instruction decryption, which does not add any observable delay. Moreover, the modified kernel performs minor extra tasks: it reads the key from the executable file (for static encryption) or it randomly generates a new key (for dynamic encryption) only once per each execution, while it adds just 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 slight slowdown of 1%–1.2% in static and 1%–1.5% in dynamic encryption. This deviation is probably due to the different linking configurations (statically linked versus dynamically linked shared libraries).

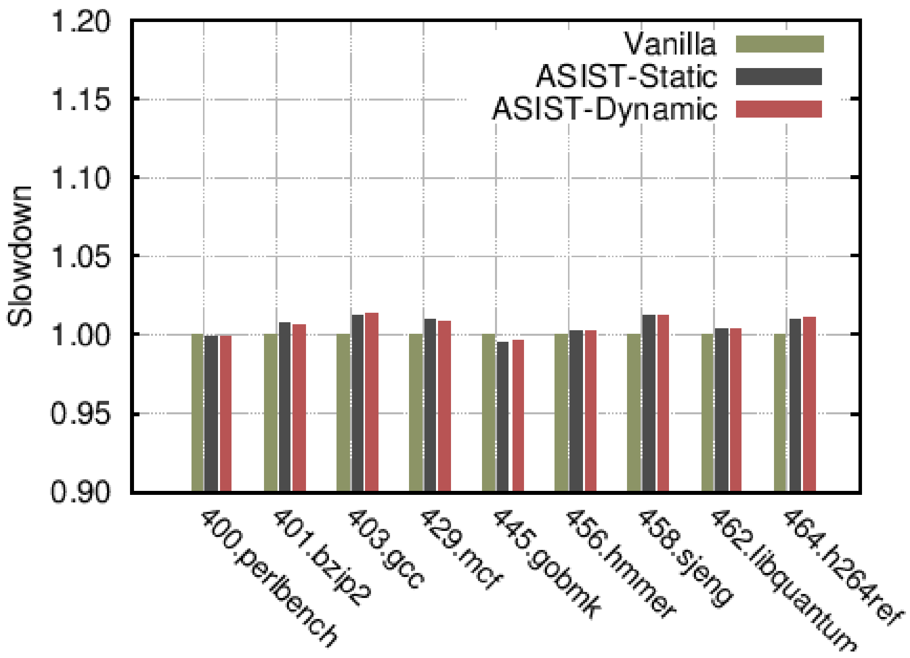


Figure 5.1: *Runtime overhead using the SPEC CPU2006 benchmark suite.* We see that both ASIST implementations have negligible runtime overhead compared to the vanilla system.

Benchmark	**Data page faults per second**	**Text page faults per second**
400.perlbench	38.4964	1.97215
401.bzip2	44.3605	0.193831
403.gcc	60.3235	3.93358
429.mcf	51.7769	0.0497679
445.gobmk	25.4735	0.905984
456.hmmer	0.0546246	0.0223249
458.sjeng	71.9751	0.0676988
462.libquantum	5.18675	0.0486765
464.h264ref	3.19614	0.0333707

Table 5.2: *Data and text page faults per second when running the SPEC CPU2006 benchmark suite.* All benchmarks have very few text page faults per second, which explains the negligible overhead of the dynamic encryption approach.

Setup	**Slowdown**
Vanilla	1.0
ASIST-Static	1.0026
ASIST-Dynamic	1.0035

Table 5.3: *Slowdown when inserting data into an sqlite3 database.* We see that ASIST achieves very close runtime performance with the vanilla system.

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 5.1 indicate that dynamic encryption performs equally well with static encryption. Thus, our proposed approach to dynamically encrypt program code at the page fault handler, instead of statically encrypt the code before program's execution, does not seem to add any extra overhead.

To better understand the performance of this approach, we instrumented the Linux kernel to measure the data and text page faults of each process that uses the dynamic encryption mode. Table 5.2 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. Therefore, we notice a negligible overhead with dynamic code encryption at the page fault handler for two main reasons: (i) as we see in Table 5.2, 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.

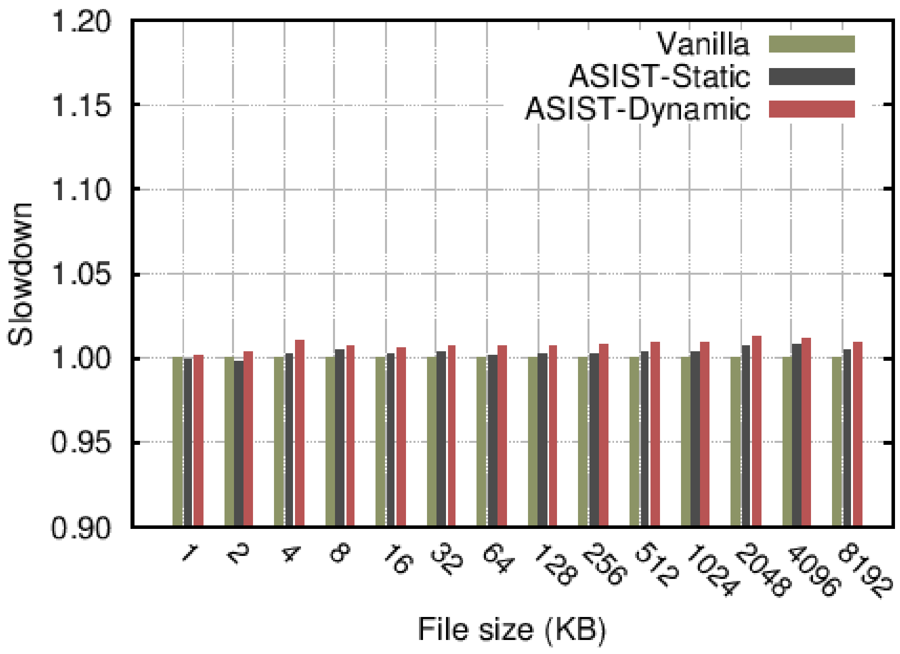


Figure 5.2: *Slowdown when downloading different files from a lighttpd Web server as a function of the file size.* We see that ASIST adds less than 1% delay for all file sizes.

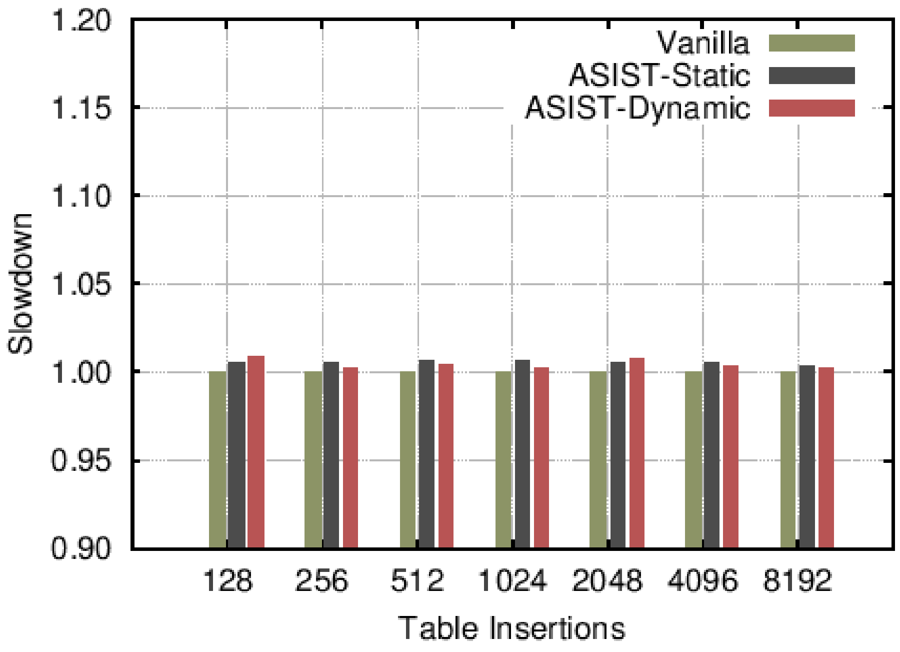


Figure 5.3: *Slowdown when inserting data into sqlite3 as a function of the number of insertions.* We see that ASIST experiences less than 1% slowdown even for very small datasets.

5.2.2 Real-world Applications

We evaluated ASIST with two real-world applications. First, we ran the `lighttpd` Web server in a vanilla system and in the two versions of ASIST. We used another machine located in the local network to repeatedly download 14 files of different sizes, ranging from 1 KB to 8 MB, and we measured the average download time for each file. Figure 5.2 shows the slowdown of the download time as a function of the file size for each system. We see that ASIST does not impose any considerable delay, as the download time remains within 1% of the vanilla system for all files. We also notice that both static and dynamic encryption implementations perform almost equally good. We measured the page faults caused by `lighttpd`: 261 data page faults per second, and just 0.013 text page fault per second. Thus, the dynamic encryption did not add any runtime overhead to the server. Moreover, we observed that most of the text page faults occur during the first few milliseconds of the `lighttpd` execution, when the code is loaded into memory, and then practically no text page fault occurs.

In our last benchmark we ran an `sqlite3` database in the vanilla and in the two ASIST setups. To evaluate the performance of `sqlite3` we used the C/C++ SQLite interface to implement a simple benchmark that reads a large tab-separated file and updates a table's entries with the respective values in an aggressive manner. Table 5.3 shows the results of this benchmark that measures data insertions and selections Figure 5.3 shows the slowdown when inserting data into the database using this benchmark as a function of the number of insertions for all setups. We observe that as in all previous cases, ASIST imposes less than 1% slowdown on the database's operation for both static and dynamic approaches, even on small datasets that do not provide ASIST with enough time to amortize the encryption overhead.

5.3 Memory Overhead

In order to successfuly encrypt the pages containing the executable code in the dynamic implementation we create an anonymous copy of the mapped page (*copy on write*). The fact that we create a new page per code/text page that we see in each page fault, means that resident memory of the binary due to the text page faults will be twice as much comparing it with the Vanilla or static implementation. Although double the memory footprint of a running process may seem like a lot, that is not entirely true. We should keep in mind that the text pages caused by the text page faults are not brought into memory all at once, instead they are scattered throughout the program's execution, thus diminishing the memory footprint of the running process.

Benchmark	**Vanilla Tata PFs**	**ASIST-Static Text PFs**	**ASIST-Dynamic Text PFs**
400.perlbench	3417	3458	3416
401.bzip2	6673	6638	66375
403.gcc	8924	8868	8871
429.mcf	19768	19810	19767
445.gobmk	4849	4807	3911
456.hmmer	161	117	116
458.sjeng	44653	44691	44458
462.libquantum	2878	2921	2877
464.h264ref	9909	9866	9832

Table 5.4: *We see the Text page faults generated by the operating system when running the SPEC in all 3 implementation.* As it appears from the numbers, there is no significant difference in the number of the generated page faults between the Vanilla implementation and the other two ASIST implementations

5.3.1 External Fragmentation

Due to the rules governing memory allocation, more computer memory is sometimes allocated than is needed mostly due to data alignment. Unlike other types of fragmentation, internal fragmentation is difficult to reclaim and that is why programms usually live with it. External fragmentation on the other way arises when free memory is separated into small blocks/pages and is interspersed by allocated memory. The problem of external fragmentation is mostly handled by compiler optimizations which by intermixing data and code and thus utilizing a page size to its fullest.

As we describe in Section 3.2 our scheme requires that data and code need to be seperated. Thus, we modified the linker in order to produce a binary which ELF format wouldn't allow intermixed code and data. We expected that this modification would have an impact on the number of the generated text and data faults. In Table 5.4 we present the number of only the text faults generated throught the execution of the SPEC benchmarks in all three implementations. As we can see from the Table 5.4, most of the benchmarks have a negligible increase in the number of the generated text faults, while in some cases, e.g.hmmer the vanilla implementation seems to generate more text faults than both the static and dynamic implementations.

6
Related Work

6.1 Instruction Set Randomization

ISR was initially introduced as a generic defense against code injections by Kc et al. [33] and Barrantes et al. [11, 12]. To demonstrate the feasibility of ISR, they proposed implementations in `Bochs` [35] and `Valgrind` [38] respectively. Hu et al. [31] implemented ISR with Strata SDT tool [45] using AES as a stronger encryption for instruction randomization. Boyd et al. [14] proposed a selective ISR to reduce the runtime overhead. Portokalidis and Keromytis [44] implemented ISR using Pin [36] with moderate overhead and shared libraries support. In Section 2.3 we described in more detail all the existing software-based ISR implementations and we compared them with ASIST. ASIST addresses most of the limitations of the existing ISR approaches owing to its simple and efficient hardware and operating system support.

6.2 Other Defenses against Code Injection Attacks

Modern hardware platforms support non-executable data protection, such as the No eXecute (NX) bit [41]. This feature prevents stack or heap data from being executed by marking the respective memory pages with the NX bit, so it is capable to protect systems against code injection attacks without any 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. This may be due to backwards compatibility constraints, self-modifying code, or bad programming practices.Also, the NX bit cannot prevent *return to libc* attacks that use existing code to accomplish the attack [48]. We believe that ASIST can be used complementary to NX bit to serve as an additional layer of security, e.g., 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 the pages that contain injected code executable, and so bypassing the NX bit protection mechanism. This way, they can execute arbitrary code to implement the attack without the need to identify more specific gadgets, which may not be easy to find, e.g., due to the use of Address Space Layout Randomization (ASLR). In contrast, these exploits cannot execute any injected code in a system using ASIST, as this code will not be correctly encrypted and will not be successfully executed in the ASIST processor. In such cases, ISR as an inherent part of the system, as suggested by ASIST, is able to offer another security layer by transparently protecting against any type of code injection attacks. Thus, ASIST with ASLR provides a stronger defense against such attacks.

A recent attack demonstrated by Snow et al. [50] is also able to bypass NX bit and ASLR protection 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. ASIST 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. ASIST ensures that key does not reside in process's memory, while stronger encryption algorithms (like AES) can also fit in our design to avoid inferring the key.

SecVisor [46] protects the kernel from code injection attacks using a hypervisor to prevent unauthorized code execution in the kernel. While SecVisor focuses on kernel's code integrity, ASIST prevents the execution of unauthorized code in both user- and kernel-level by implementing efficiently and transparently ISR with hardware support.

6.3 Defenses against Buffer Overflow Attacks.

A significant number of research efforts have been made to provide protection against buffer overflow attacks. StackGuard [22] uses canaries to protect the stack, while PointGuard [21] protects function pointers from buffer overflows by encrypting 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 [49] protects existing binaries from buffer overflows without access to source code, by discovering the data structures and then rewriting the binary.

6.4 Other Randomization-based Defenses.

Address Space Layout Randomization (ASLR) [42] randomizes the memory layout of a process at runtime or at compile time to protect against code-reuse attacks. Giuffrida et al. [29] propose an approach with address space randomization to protect the operating system kernel. Bhatkar et al. [13]

present randomization techniques for the addresses of the stack, heap, dynamic libraries, routines and static data in an executable. Wartell et al. [56] randomize the instruction addresses at each execution to address code-reuse attacks. Jiang et al. [32] prevent code injections by randomizing the system call numbers.

6.5 Hardware Support for Security

There are numerous research efforts aiming to provide hardware support for security without sacrificing performance. Dalton et al. [23, 24] propose a hardware-based architecture for dynamic information flow tracking, by extending a SPARC V8 processor with four tag bits per each register and memory word, as well as with tag propagation and runtime checks to defend against buffer overflows and high-level attacks. Greathouse et al. [30] present a design for accelerating dynamic analysis techniques with hardware support for unlimited watchpoints. These efforts significantly reduce the performance cost for dynamic information flow analysis, which has a very high overhead in software-based implementations. Frantzen and Shuey [27] implement a hardware-assisted technique for the SPARC architecture to provide return address protection. Tuck et al. [54] propose hardware encryption to protect function pointers from buffer overflow attacks with improved performance, extending the computationally expensive software-based pointer encryption used by pointguard [21]. Our approach is similar to these works: we also propose hardware support for another existing technique that prevents the execution of any code that is not authorized to run in the system.

7 Conclusion

We have presented the design, implementation and evaluation of ASIST: a hardware-assisted architecture for ISR support. ASIST is designed to offer (i) improved performance, without runtime overhead, (ii) improved security, by protecting the operating system and resisting key guessing attempts, and (iii) transparent operation, with shared libraries support and no need for any program modifications. Our experimental evaluation shows that ASIST does not impose any significant overhead (less than 1.5%), while it is able to prevent code injection attacks that exploit user-level and kernel-level vulnerabilities. We have also proposed a new approach for dynamic code encryption at the page fault handler when code is first loaded into process' memory. This approach transparently encrypts unmodified binaries that may use shared libraries with a new key at each execution, offering protection against incremental key guessing attacks. Our results indicate that dynamic code encryption is efficient, without adding any overhead due to the low text page fault rate. Our work shows that ASIST can address most of the limitations of existing software-based ISR implementations while adding less than 0.7% additional hardware to a SPARC processor. We believe that ASIST can be easily ported to other architectures to strengthen existing defenses against code injection attacks.

Bibliography

[1] Common vulnerabilities and exposures (cve). http://cve.mitre.org/.

[2] The metasploit project. http://www.metasploit.com/.

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

[4] USA National Vulnerability Database. http://web.nvd.nist.gov/view/vuln/statistics.

[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 Security Bulletin MS08-067 – Critical. http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx, 2008.

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

[9] Microsoft security advisory (975191): Vulnerabilities in the ftp service in internet information services. http://www.microsoft.com/technet/security/advisory/975191.mspx, 2009.

[10] Microsoft security advisory (975497): Vulnerabilities in smb could allow remote code execution. http://www.microsoft.com/technet/security/advisory/975497.mspx, 2009.

[11] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized Instruction Set Emulation. *ACM Transactions on Information and System Security*, 8(1), 2005.

[12] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

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

[14] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis. On the General Applicability of Instruction-Set Randomization. *IEEE Transactions on Dependable Secure Computing*, 7(3), 2010.

[15] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[16] P. P. Bungale and C.-K. Luk. PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation. In *ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2007.

[17] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems. In *Asia-Pacific Workshop on Systems (APSys)*, 2011.

[18] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, 2005.

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

[20] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *USENIX Security Symposium*, 2001.

[21] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointguardTM: Protecting Pointers from Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*, 2003.

[22] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[23] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information flow Architecture for Software Security. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2007.

[24] M. Dalton, H. Kannan, and C. Kozyrakis. Real-World Buffer Overflow Protection for Userspace & Kernelspace. In *USENIX Security Symposium*, 2008.

[25] D. Danchev. Managed polymorphic script obfuscation services. `http://ddanchev.blogspot.com/2009/08/managed-polymorphic-script-obfuscation.html`, 2009.

[26] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in c/c++. In *International Conference on Software Engineering (ICSE)*, 2012.

[27] M. Frantzen and M. Shuey. StackGhost: Hardware Facilitated Stack Protection. In *USENIX Security Symposium*, 2001.

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

[29] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *USENIX Security Symposium*, 2012.

[30] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin. A Case for Unlimited Watchpoints. In *ACM Intrnational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[31] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and Practical Defense Against Code-Injection Attacks using Software Dynamic Translation. In *ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2006.

[32] X. Jiang, H. J. Wangz, D. Xu, and Y.-M. Wang. RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2007.

[33] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[34] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection Against Return-to-User Attacks. In *USENIX Security Symposium*, 2012.

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

[36] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

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

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

[39] J. Oberheide, M. Bailey, and F. Jahanian. PolyPack: An Automated Online Packing Service for Optimal Antivirus Evasion. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[40] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis. Asist: architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, CCS '13, pages 981–992, New York, NY, USA, 2013. ACM.

[41] L. D. Paulson. New Chips Stop Buffer Overflow Attacks. *IEEE Computer*, 37(10), 2004.

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

[43] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C analysis. *SRI International*, 2009.

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

[45] 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.

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

[47] S. Sethumadhavan, S. J. Stolfo, A. Keromytis, J. Yang, and D. August. The SPARCHS Project: Hardware Support for Software Security. In *SysSec Workshop*, 2011.

[48] H. 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.

[49] A. Slowinska, T. Stancescu, and H. Bos. Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation. In *USENIX Annual Technical Conference (ATC)*, 2012.

[50] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy*, 2013.

[51] A. N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? the Effectiveness of Instruction Set Randomization. In *USENIX Security Symposium*, 2005.

[52] Standard Performance Evaluation Corporation (SPEC). SPEC CINT2006 Benchmarks. `http://www.spec.org/cpu2006/CINT2006`.

[53] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the Memory Secrecy Assumption. In *ACM European Workshop on System Security (EUROSEC)*, 2009.

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

[55] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving Integer Security for Systems with KINT. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2012.

[56] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[57] Y. Weiss and E. G. Barrantes. Known/Chosen Key Attacks against Software Instruction Set Randomization. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

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