On Architectural Support for Instruction Set Randomization

GEORGE CHRISTOU, GIORGOS VASILIADIS, and VASSILIS PAPAEFSTATHIOU,
Foundation for Research and Technology Hellas (FORTH-ICS), Greece
ANTONIS PAPADOGIANNAKIS, Unaffiliated
SOTIRIS IOANNIDIS, Technical University of Crete (TUC-ECE), Greece

Instruction Set Randomization (ISR) is able to protect against remote code injection attacks by randomizing the instruction set of each process. Thereby, even if an attacker succeeds to inject code, it will fail to execute on the randomized processor. The majority of existing ISR implementations is based on emulators and binary instrumentation tools that unfortunately: (i) incur significant runtime performance overheads, (ii) limit the ease of deployment, (iii) cannot protect the underlying operating system kernel, and (iv) are vulnerable to evasion attempts that bypass the ISR protection itself.

To address these issues, we present the design and implementation of ASIST, an architecture with both hardware and operating system support for ISR. ASIST uses our extended SPARC processor that is mapped onto a FPGA board and runs our modified Linux kernel to support the new features. In particular, before executing a new user-level process, the operating system loads its randomization key into a newly defined register, and the modified processor decodes the process's instructions with this key. Besides that, ASIST uses a separate randomization key for the operating system to protect the base system against attacks that exploit kernel vulnerabilities to run arbitrary code with elevated privileges. Our evaluation shows that ASIST can transparently protect both user-land applications and the operating system kernel from code injection and code reuse attacks, with about 1.5% runtime overhead when using simple encryption schemes, such as XOR and Transposition; more secure ciphers, such as AES, even though they are much more complicated for mapping them to hardware, they are still within acceptable margins,with approximately 10% runtime overhead, when efficiently leveraging the spatial locality of code through modern instruction cache configurations.

CCS Concepts: • **Security and privacy → Embedded systems security**; *Hardware-based security protocols*;

Additional Key Words and Phrases: Code injection, instruction set randomization, hardware assisted security

Work performed while he was at FORTH-ICS.

This article is an extended version of the conference paper [47] presented at CCS'13 and provides improvements in terms of security (i.e., hardware implementation of the AES cryptographic algorithm), performance (i.e., increased cache locality due to the decryption of instructions before the instruction cache), and portability (i.e., blockwise ISA-agnostic decryption of instructions). This work is supported by the Europian European Commission under the Horizon 2020 Program through the CONCORDIA project (Grant Agreement No. 830927), C4IIoT project (Grant Agreement No. 833828), COLLABS (Grant Agreement No. 871518), and RESIST project (Grant Agreement No. 769066). The authors thank Nikolaos Dimou for his invaluable comments and suggestions during the implementation of the ASIST prototype.

Authors' addresses: G. Christou, G. Vasiliadis, and V. Papaefstathiou, Foundation for Research and Technology Hellas (FORTH-ICS), Heraklion, Greece; emails: {gchri, gvasil, papaef}@ics.forth.gr; A. Papadogiannakis, Unaffiliated; email: apapadog@gmail.com; S. Ioannidis, Technical University of Crete (TUC-ECE), Chania, Greece; email: sotiris@ece.tuc.gr.

1544-3566/2020/11-ART36
https://doi.org/10.1145/3419841

ACM Reference format:
George Christou, Giorgos Vasiliadis, Vassilis Papaefstathiou, Antonis Papadogiannakis, and Sotiris Ioannidis. 2020. On Architectural Support for Instruction Set Randomization. *ACM Trans. Archit. Code Optim.* 17, 4, Article 36 (November 2020), 26 pages.
https://doi.org/10.1145/3419841

1 INTRODUCTION

Code injection attacks enables an attacker to execute malicious code through the exploitation of a software vulnerability. The vast majority of code injection attacks exploit vulnerabilities that allow the diversion of control flow to the injected malicious code. Arbitrary code execution is also possible through the modification of non-control-data [20]. The most commonly exploited vulnerabilities for code injection attacks are buffer overflows [2]. Despite considerable research efforts [25, 26, 62], buffer overflow vulnerabilities remain a major security threat [21]. Other vulnerabilities that allow the corruption of critical data are format-string errors [23] and integer overflows [63].

Instruction set randomization (ISR) has been proposed as a countermeasure against code injection attacks [8, 37, 50]. ISR randomizes the instruction set (ISA) of a processor so that an attacker is not able to know the instruction set of the target machine to inject or disclose 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 typically encrypt the instructions of a program with a specific key. This key actually defines the valid instruction set for this program. The processor decrypts at runtime every instruction of the respective process with the same key. Only the correctly encrypted instructions will lead to the intended code after decryption. Any injected code that is not encrypted with the correct key will result in irrelevant or invalid instructions. At the same time, code fragments, called gadgets, cannot be disclosed in a decrypted form, hence not chained into a meaningful attack payload (as it typically happens in code-reuse attacks such as ROP [17] and JOP [14]).

Most of the existing ISR implementations utilize binary transformation tools in order to encrypt the text section of the target binaries. Running those binaries requires an emulator that is responsible for decrypting the instructions at runtime [16, 37]. Other solutions rely on dynamic binary instrumentation tools [35, 50]. Such approaches have several limitations though: (i) They incur a significant runtime performance overhead due to the software emulator or instrumentation tool. This overhead is often prohibitive for the wide adoption of such techniques. (ii) Deployment is limited by the necessity of several tools, like emulators, simulators, and manual encryption of the programs that are protected with ISR. (iii) They are vulnerable to code injection attacks into the underlying emulator or instrumentation tools. More importantly, they do not protect against attacks targeting kernel vulnerabilities [3–5, 19], which are becoming an increasingly attractive target for attackers. (iv) Most ISR implementations are vulnerable to evasion attacks aiming to guess the encryption key and bypass ISR protection [60, 65].

To address these issues, we propose ASIST: a hardware/software scheme to support ISR on top of an unmodified ISA. Similarly to other proposed hardware extensions that enhance security [26, 34, 54, 62], we advocate that hardware support for ISR is essential to guard against user- and kernel-level code injection attacks, while the performance penalty incured is within acceptable margins. To randomize the code instructions, we implement three different encryption algorithms, tailored for different needs and usages: (i) XOR, (ii) Transposition, and (iii) AES. As we have shown in our previous work [47], XOR and Transposition require minimal resources in the processor die; however, they are vulnerable to key guessing attacks and cryptanalysis. To overcome this, in this article we extend our previous implementation with more secure ciphers, such as AES. The

AES algorithm requires significantly more modifications on the processor architecture; however, it offers enhanced protection compared to XOR and Transposition.

In addition, we explore two possible choices for implementing runtime decryption at the CPU: (i) at the instruction fetch pipeline stage of the modified processor and (ii) between the MMU and the instruction cache. We compare two techniques for encrypting the executable code: (i) statically, by adding a new section in ELF that contains the key that has been used to encrypt all code sections of the binary file, using a binary transformation tool, and (ii) dynamically, by generating a random key at load time and encrypting with this key at the page fault handler all the executable memory mapped pages. The dynamic encryption approach supports dynamically linked shared libraries, whereas static encryption requires statically linked binaries. We discuss and evaluate the advantages of each approach in terms of security and performance. Finally, our modified processor can also encrypt the return address at each function call and decrypt it right before returning to the caller. By doing so, ASIST is capable to protect against ROP attacks by (i) preventing gadget discovery, via the use of strong encryption schemes (i.e., AES) [17, 55], and by (ii) limiting the discovery of call-preceded code locations, via the encryption of the return addresses [18].

To demonstrate the feasibility of our approach, we present the prototype implementation of ASIST by modifying the Leon3 SPARC V8 processor [1], a 32-bit open-source synthesizable processor [28]. 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 practically without any performance overhead, i.e., less than 1%, when using simple encryption schemes such as XOR and Transposition; more secure ciphers, such as AES, introduce a slightly higher overhead, about 10%, which are acceptable in real scenarios considering the benefits in terms of security. Meanwhile the hardware extensions add about 10% of additional hardware to support all ISR modes of our design. Our results also indicate that the dynamic code encryption at the page fault handler does not impose significant overhead, due to the low page fault rate for executable pages. This outcome makes our dynamic encryption approach very appealing, as it is able to generate a different, random key at each execution, *transparently* encrypt any executable program, and support shared libraries with negligible overhead.

Overall, the main contributions of our work are as follows:

- We design and implement ASIST, the first, to the best of our knowledge, hardware-based support for Instruction-Set Randomization (ISR). Our evaluation results show that a hardware-based ISR implementation, like ASIST, is able to prevent code injection attacks and protect the system against attacks that exploit OS kernel vulnerabilities, at negligible overhead.
- We extend our previous work, which was based on simple XOR and Transposition techniques, by adding the AES cipher. By doing so, we can guarantee that an encryption key cannot be derived even if the attacker has access to both the plaintext and the ciphertext (e.g due to a memory leak). In addition, it can hinder any gadget discovery—which actually is a pivotal step of code reuse attacks—that are based in code pointer leaks to bypass ASLR and the exploitation of memory disclosure vulnerabilities to map the text segment of a process; in both cases instructions will be encrypted with a strong cryptography scheme that prevents any form of cryptanalysis or bruteforce attacks.
- To enable AES, we replicated the cache line fill protocol and placed the ASIST unit between the MMU and the instruction cache. This new MMU component is responsible for decrypting the instructions blockwise before sending them to the cache subsystem. This is a major contribution over the previous design, in which the instructions where decrypted only after they were fetched from the cache, for several reasons: First, it increases the performance

due to spatial locality. Second, the new component makes our design ISA-agnostic, hence more easily portable to other architectures, such as CISC.
- We introduce a dynamic code encryption technique that can transparently encrypt pages with executable code at the page fault handler, using a randomly generated key for each execution. This technique can support shared libraries and does not impose significant overhead.
- In terms of performance evaluation, we performed extra experiments to see how the instruction cache size affects the performance of ISR, showing that when we increase the cache size (to 32 KB or 64 KB) the runtime overhead of a multi-round, block, cipher, such as AES is reduced to practical margins. Since we evaluate our design using an FPGA we also offer measurements regarding the area overhead.

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

Our threat model includes any programming bugs or inadvertent design flaws in software that can be exploited by an active adversary to launch a code injection or code reuse attack. The adversary does not have administrative access (i.e., superuser) and can have either local or remote access to the system. We also consider a trusted operating system and that the executable files cannot be accessed when stored on disk. The operating system is also protected by our design against code reuse and code injection attacks. Any hardware bugs or vulnerabilities that may arise due to hardware design choices (such as Spectre [39] and Meltdown [43]) are outside of our threat model, as mitigation would require orthogonal solutions for hardware reliability. Finally, we do not target denial-of-service attacks or side-channel attacks. Below, we list different type of attacks and describe how ASIST can protect in each case.

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

Kernel vulnerabilities. Remotely exploitable vulnerabilities on the operating system kernel [3–5, 19] are becoming an increasingly attractive target for attackers. Our threat model includes code injection and code reuse attacks based on kernel vulnerabilities. We propose an architecture that is capable of protecting the operating system kernel as well. Our design can also thwart attacks that use a kernel vulnerability to run user-level code with elevated kernel privileges [38].

Return-to-libc and Code Reuse Attacks. Instead of injecting new code into a vulnerable program, an attacker can execute existing code upon changing the control flow of a vulnerable system to perform the attack. This can be done either by redirecting the execution to existing library functions, attacks typically known as *return-to-libc* attacks [45]; either by using existing instruction sequences ending with a *ret* instruction (called *gadgets*), a technique known as *return-oriented programming* (ROP) [17, 55]. These types of attacks can be also achieved by chaining gadgets with function pointers residing in the stack [15]. ISR was originally designed to protect a system against code injection attacks, and not to address return-to-libc and ROP attacks [37].

However, ISR with strong encryption can prevent the discovery of usable gadgets, which is the pivotal step of code reuse exploitation techniques. In the case where an attacker has managed to disclose the text pages of a process: the text pages will be encrypted using a different key for each process, prohibiting any gadget extraction (e.g., though cryptanalysis or bruteforce attacks). Snow et al. [59] introduced the concept of Just-In-Time-ROP attacks, in which even a single leak of a code pointer and the presence of a memory disclosure vulnerability can be exploited by an attacker to map the text segment of a process. This can be achieved by recursively scanning disclosed code pages for more code pointers. A similar technique described by Bittau et al. [12] can circumvent ASLR. Herein, an attacker will exploit available memory disclosure vulnerabilities to bypass ASLR. Then, brute force will be used in order to find a ROP sequence that can read the code pointed from the leaked code pointers and send them to the attacker (e.g., write to socket). The attack has been demonstrated to complete within approximately 4,000 requests. Again, ISR makes it hard for an attacker to discover usable gadgets, since the text segments will be encrypted.

Key guessing attacks. Existing ISR implementations are vulnerable to key guessing or key stealing attacks [60, 65]. This way, sophisticated attackers may be able to bypass the ISR protection mechanism, by guessing the key and then injecting and executing code that is correctly encoded with this key. In this work, we aim to design and implement ISR in a way that it will be very difficult for attackers to guess or infer the code randomization key. To achieve this goal, we extend our previous work [47], which was based on simple XOR and transpotition techniques, by implementing AES; by doing so, an attacker with knowledge of both the plain-text and cipher-text of instructions will not be able to reconstruct the encryption key.

Transient execution attacks. Any vulnerabilities in hardware that can disclose protected or inaccessible memory when successfully exploited, cannot be mitigated by ASIST—by being able to dump the whole memory, an attacker is able to retrieve the process key and circumvent our mechanism. For instance, in the recently discovered Spectre [39] and Meltdown [43] attacks, the instructions that are executed either speculatively or out-of-order can cause cache modifications. An attacker that can control the cache and observe the effects of the transiently executed instructions is capable to disclose otherwise inaccessible memory and retrieve the process key that is used for code encryption. Our mechanism is not designed to protect against such attacks, since they rely on architectural choices rather than software bugs.

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 [8]. To apply the ISR idea, existing implementations first encrypt the binary code of each program with the program's secret key before it is loaded for execution. The program's key defines the mapping of the encrypted instructions to the real instructions supported by the CPU. Then, at runtime, the randomized processor decrypts every instruction with the proper program's key before execution. Injected instruction that have not been correctly encrypted will result in irrelevant or invalid instructions after the obligatory decryption. However, correctly encrypted code will be decrypted and executed normally.

2.3 Limitations of Existing Implementations

Existing ISR Implementations use binary transformation tools, such as `objcopy`, to encrypt the code of user-level programs. For runtime decryption they use emulators [40] or dynamic binary instrumentation [44, 46, 52]. In Table 1, we list and compare existing ISR implementations.

Table 1. Comparison of ASIST with Existing ISR Implementations

ISR Implementation	Runtime Overhead	Shared Libraries	Self modifying Code	Hardware Support	Encryption	Dynamic Encryption	Kernel Protection	ROP Prevention
Bochs emulator [37]	High	No	No	No	XOR with 32-bit key	No	No	No
Valgrind tool [8, 9]	High	Yes	API	No	XOR with random key	Yes	No	No
Strata SDT [35]	Medium	No	No	No	AES with 128-bit key	No	No	No
EMUrand emulator [16]	Medium	No	No	No	XOR with 32-bit key	No	No	No
Pin tool [50]	Medium	Yes	Partially	No	XOR with 16-bit key	No	No	No
Polyglot [56]	Low to medium	Yes	Yes	Yes	AES with 128-bit key	No	Yes	Partial
Shuffler [66]	Medium	No	No	No	XOR only for return address	Yes	No	Yes
Morpheus [29]	Low[1]	No	No	Simulation	QARMA 64-bit blocks with 128-bit key	Yes	No	Yes
ARM Pointer Authentication	Medium	Yes	Yes	Only pointers	Yes	No	Yes	Yes
ASIST	Zero, Low to medium in AES	Yes	API	Yes	XOR with 32-bit–128-bit key, Transposition with 160-bit key, AES with 128-bit key	Yes	Yes	Yes

ASIST provides a hardware-based implementation of ISR without runtime overhead, it supports the necessary features of current systems and protects against kernel vulnerabilities.

Kc et al. [37] implemented ISR by modifying the `Bochs` emulator using XOR with a 32-bit key in their prototype. The use of an emulator results in significant slowdown, up to 290 times slower execution on CPU intensive applications. Barrantes et al. [8, 9] use `Valgrind` [46] to decrypt applications' code, which is encrypted with XOR and a random key equal to the program's length. This prototype supports shared libraries by copying each randomized library per process, and offers an API for self-modifying code. However, the performance overhead with Valgrind is also very high, up to 2.9 times slower than native execution. Hu et al. [35] implemented ISR with a software dynamic translation tool [52] using AES encryption with 128-bit key size. Dynamic translation results in lower but still significant performance overhead that is close to 17% on average and as high as 250%. To reduce runtime overhead, Boyd et al. [16] 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 [50] implemented ISR with shared libraries support using Pin [44]. The runtime overhead ranges from 10% to 75% for popular applications, while it has four-times slower execution when memory protection is applied to Pin's code.

Polyglot [56] is similar with ASIST but does not offer dynamic encryption. Thus, attackers can still launch blind ROP attacks, since the code will not be encrypted with different keys on each launch. Moreover, it uses a separate shared key cache inside the processor, in order to support dynamic libraries, which increases the circuit area overhead of the proposed mechanism significantly. Shuffler [66] proposes a mechanism that periodically re-randomizes the address space layout of a process. The proposed mechanism offers adequate protection against control-flow attacks by continuously modifying the code pointer addresses. However, it is not scalable, because it requires a parallel thread to each protected process, responsible for the re-randomization. Moreover, the proposed mechanism can protect only user-level applications. Morpheus [29] is a promising architecture for thwarting the majority of control-flow attacks. Yet, it has been only implemented

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

and evaluated on gem5 simulator [11], with a single process context, using the system call emulation capabilities of gem5, without any estimation of the area overhead of their design. The proposed mechanism requires extensive modifications of the original processor by adding several hardware components, which may be impractical for low-powered or resource-contraint devices (e.g., IoT devices), due to the extra substantial area overhead Finally, the authors state that the latency of their mechanism will be prominent in pipelined processors; this does not seem to be the case though, since every execution unit of an out-of-order processor should be extended with an attack detector, imposing even more area overhead in the final design. ARM recently launched processors with pointer authentication capabilities [41]. In this approach, authentication codes are created for every control-flow pointer, calculated with special instructions. If an adversary modifies pointers residing in the stack, then the modification will be detected by the authenticating instructions placed just before the control-flow transition instruction, i.e., indirect jump, return. However, Google project zero reviewed the deployment of the PAC and found that forging pointer authentication codes was possible [33].

To put our work into context, we provide a quantitatively and qualitatively comparison in Table 1. ASIST is the first ISR implementation with hardware support, resulting in negligible runtime overhead for any type of applications. ASIST also introduces 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) AES support for resilience even when memory is disclosed. Moreover, ASIST prevents the execution of injected code at the kernel by using separate keys for user-level programs and kernel's code. Finally, ASIST is able to prevent Code Reuse Attacks using return address encryption, while hindering the discovery of the code layout even when memory leaks are present.

3 ASIST ARCHITECTURE

ASIST consists of different modules and components across different layers of the hardware, the operating system, and the user-space, as can be shown in Figure 1. Overall, the processor has been extended with two new registers: *usrkey* and *oskey*, which store the keys of the running user-level process and operating system kernel's code, respectively. Additionally, a new register *asist_mode* is used to select the encryption algorithm. The operating system keeps the key and the mode of each process in a respective field in the process table and stores the key and the mode of the next process that is scheduled for execution in the *usrkey* and *asist_mode* registers using the *sta* privileged SPARC instruction. Moreover, the processor is modified to decrypt instructions before they are fetched from RAM, using one of the above two keys, according to the supervisor bit.

3.1 Encryption

ASIST offers three different algorithms for code encryption, namely XOR, Transposition, and AES. These algorithms have different performance and security characteristics, as we will see in more detail in Section 3.1.5 and Section 5.2, and can be used to cover different needs or requirements. We support two options for encrypting an executable program: static and dynamic. In static encryption, the program is encrypted before each execution with a pre-defined key. In dynamic encryption, a key is randomly generated at the binary loader, and all code pages are encrypted with this key at the page fault handler before they are mapped to the process's address space.

The main advantage of static code encryption is that it has no runtime overhead for XOR and Transpotition, while imposing acceptable overhead when using AES. However, this approach has

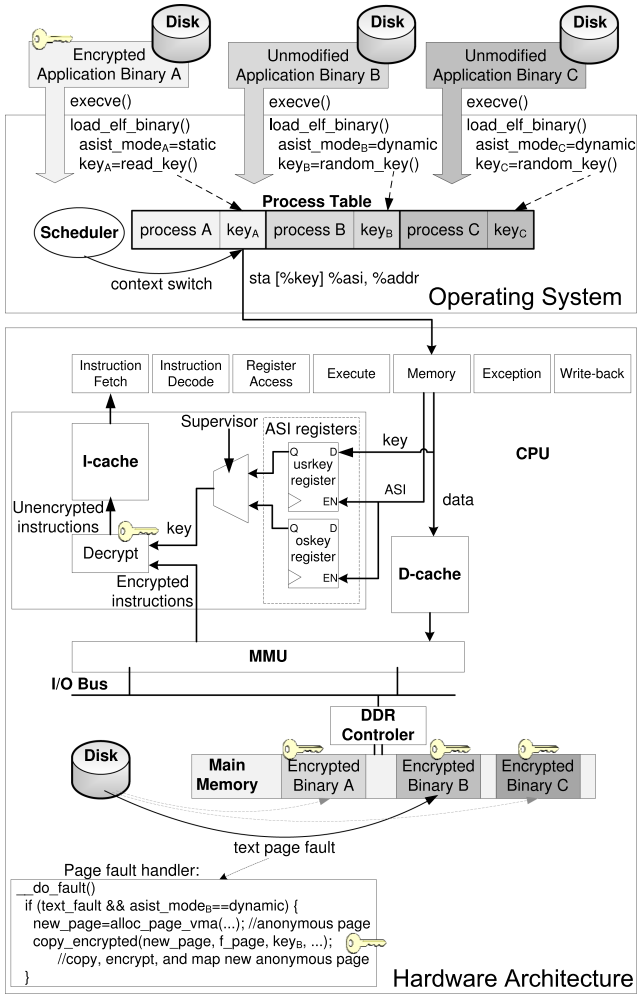


Fig. 1. ASIST architecture. The operating system reads the key from the ELF binary (static encryption) or randomly generates a new key (dynamic encryption), saves the key in the process table, and stores the key of the running process in the *usrkey* register. The processor decrypts each instruction (or I-cache line in case of AES) using *usrkey* or *oskey* registerm, according to the *supervisor* bit.

several drawbacks. First, the same key is used for each execution, which makes it susceptible to brute force attacks trying to guess this key. Second, each executable file needs to be encrypted before running. Third, static encryption does not support shared libraries; all programs must be statically linked with all necessary libraries. In contrast, dynamic encryption has a number of advantages: it generates a random key at each execution so it cannot be easily guessed, it encrypts all executables transparently without the need to run an encryption program, and it is able to support shared libraries. The drawback of dynamic encryption is the runtime overhead to encrypt a code page when it is loaded to memory at a code page fault. In Section 5, we show that due to the low number of code page faults, dynamic encryption is very efficient.

3.1.1 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 or randomly

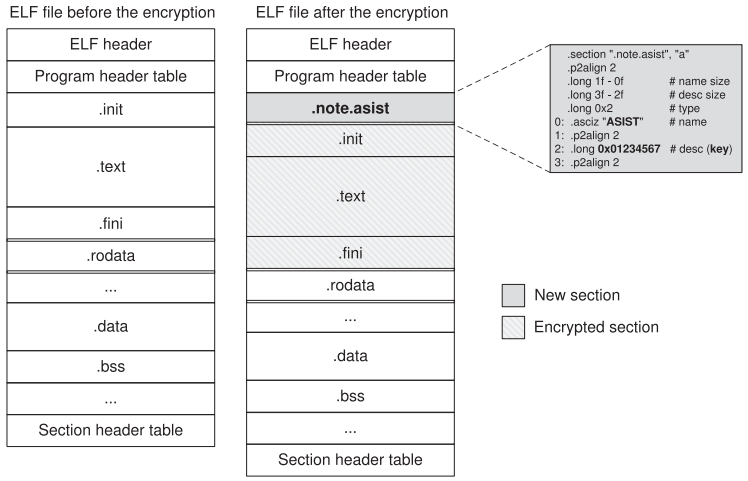


Fig. 2. The ELF format of a statically encrypted executable file. The key is stored in a new note section inside the ELF file, and all the code sections are encrypted with this key.

chosen by the tool. Figure 2 shows the modifications of a statically encrypted ELF binary. We add a new note section (`.note.asist`) inside the encrypted ELF file that contains the program's encryption key. We also changed the ELF binary loader in the Linux kernel to read the note section from the ELF, get the key, and store it in a new field (`key`) of the current process. In this operation mode, we set a new field per process (`asist_mode`) to static. The key is stored in the process table and is used by the kernel to set the *usrkey* hardware register each time this process is scheduled for execution.

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

To address the issue of using the same key at all executions, which may facilitate a key guessing attack, one approach could be to re-encrypt the binary after a process crash. 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. Even though this approach can occur considerable overheads due to the extra time needed to copy and encrypt the entire binary at load time, we believe that this can be useful to protect against attacks that probe applications continuously to find execution traces that do not crash. By re-randomizing the binary code each time, the possibilities to achieve such text segment disclosements are eradicated.

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

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

Moreover, we allocate an anonymous page, i.e., a page that is not backed by a file, and copy the encrypted code on this page, so that the changes will not be stored at the original binary file. Even though processes that run the same code could share the respective code pages in physical memory, we have a separate copy of each page with executable code for each process, as they have been encrypted with different keys. This may result in a small memory overhead, but it is necessary to use a different key per process and achieve better isolation. As shown from recent attacks [12] this approach can prevent probed processes to be forced to use the same key every time they are restarted after being crashed by the attacker.

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

Finally, we also modified the `fork()` system call to randomly generate a new key for the child process. When the modified `fork()` copies the parent process's page table, it omits copying its last layer so that the child's code pages will not be mapped with pages encrypted with the parent's key. To operate correctly, the dynamic encryption approach requires a separation of code and data per each page. For this, we modify the linker to align the ELF headers, data, and code sections to a new page by adding the proper padding.

3.1.3 Shared Libraries. One approach to support shared libraries is to bind each page with a separate key, similarly to Polyglot [56]. By doing so, shared libraries are supported using a per-page regional key. While this minimizes the memory overhead of having multiple copies of the same page for different processes, it also has several drawbacks; for example, it complicates our hardware, affects the context-switch time, and imposes additional overhead in order to fetch the key for decrypting the instructions.

To overcome this, the code of a shared library in ASIST 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 to use a different key per process, which offers better protection and isolation.

3.1.4 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. To support such programs, we added a new system call in Linux kernel, namely `asist_encrypt(char *buf, int size)`. This system call encrypts the code of length `size` that exists in the memory region starting from `buf` bytes length, using the current process's key that is stored in process table. We note that the `buf` buffer may still be vulnerable to a code injection attack, e.g., due to a buffer overflow vulnerability in the program that may lead to the injection of malicious code into `buf`. Then, this code will be correctly encrypted using `asist_encrypt()` and will be successfully executed. Like previous work supporting ISR with self-modifying code [8], we believe that programs should carefully use the `asist_encrypt()` system call to avoid malicious code injection in `buf`. By doing so, ISR can prevent even from JIT-Spraying attacks [13], where an attacker can overwrite JIT code pages residing in the heap with pages that contain malicious code; similar to code injection attacks, the attacker needs to provide encrypted code with the correct key to execute arbitrary code.

3.1.5 Encryption Algorithms and Key Size. The simplest, and probably the fastest, encryption algorithm is to XOR each bit of the code with the respective bit of the key. Since code is much larger than a typical key, the bits of the key are reused. In our prototype we implemented XOR encryption with key sizes that can range from 32 bit to 128 bit. Even though larger keys typically reduce the probability of key guesses and key extraction attacks, still they can be easily exploited when the attacker has knowledge of both the plain-text and cipher-text (e.g., due to a memory leak) [60, 65]. A better solution, with slightly more overhead, is to use the *Transposition* cipher algorithm. In Transposition the bits of a 32-bit word are shuffled using an 160-bit key. For each bit of the encrypted word we choose one of the 32 bits of the original word based on the respective bits of the key. That way, the ciphertext constitutes a permutation of the plaintext. Even though the Transposition cipher provides better security than simple XOR, it is vulnerable to anagramming.

To overcome all previous limitations, we have also implemented the AES algorithm. AES can guarantee that even attackers with the knowledge of both the plaintext and the ciphertext (e.g., due to a memory leak) cannot derive the encryption key. The implementation of a hardware-based AES for ISR though is not a trivial thing. One of the major differences of AES compared to XOR and Transposition is that it operates on 16-byte blocks. Thus, to retrieve the plain-text even of a single instruction, we need to decrypt a 16-byte aligned block. Moreover, our AES decryption unit requires approximately 12 cycles for decrypting each block instead of on-cycle decryption offered by the previous two algorithms. Other challenges are related to the specific architecture of the Leon3 processor that we used for the implementation of ASIST. For instance, the I-cache communicates with the memory interface using a 32-bit AHB bus, and hence the cache lines fill at a rate of one instruction per cycle. Moreover, Leon3 uses a critical word first policy, i.e., during a cache miss the instruction that caused the miss will be fetched first instead of the first instruction of the cache line. Then it will be immediately forwarded to both the integer unit and the I-cache. After that the rest of the cache line will be filled. To extend the previous design of ASIST [47] to support the AES algorithm, we had to make significant modifications. As we will see in Section 5.2, even when using a complex, multi-round, symmetric encryption algorithm like AES, the impact on the performance is within acceptable margins when configuring our prototype with a typical I-cache size. This is due to the implementation of a dedicated AES decryption unit placed before the I-cache. Thus, we can benefit from the I-cache locality to minimize instruction decryptions.

3.2 Hardware Support

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

When the decryption unit is after the I-cache, the instructions remain encrypted inside the cache and the decryption takes place every fetch cycle, as shown in Figure 4(b). This may add extra delays and increased power consumption—especially for complex cipher algorithms such as AES—since it is located in the critical path of the processor. Moreover, since the instructions are encrypted in the I-cache, it may jeopardize any pre-decoding operations located after the instruction cache, such as trace cache [51]. If it is stored encrypted in the I-cache, then pre-decoding cannot be performed, which may decrease performance. Some pre-decoding may be quite involving (e.g., Trace Cache would combine instructions from multiple basic blocks into a large basic block).

When the decryption unit is located before the I-cache, it is accessed only on I-cache misses, as shown in Figure 4(a). Especially for AES decryption, this is essential, since we need to decrypt a whole block (16-bytes) of instructions each time. Decrypting 16-byte blocks at once can lead to reduced power consumption, as the instructions that are executed frequently, e.g., in loops,

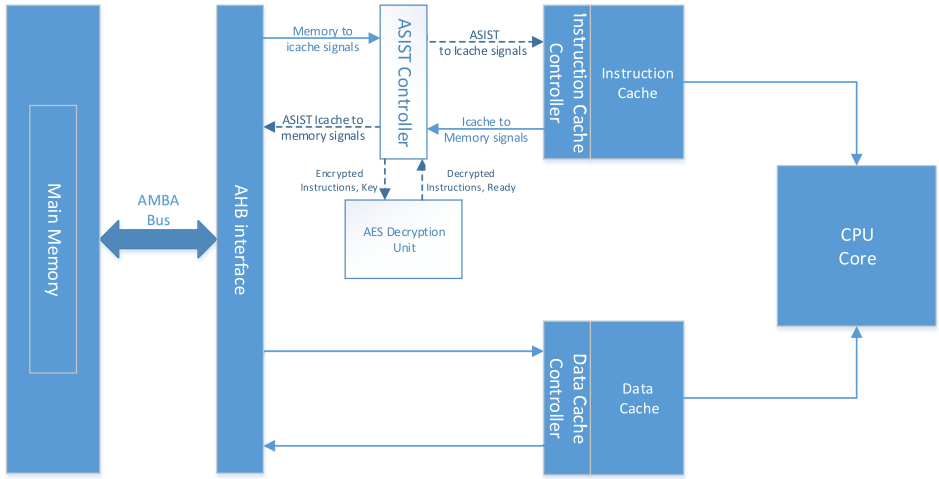


Fig. 3. ASIST hardware support for runtime instruction decryption. The AES extended ASIST processor is between the AHB interface and the I-cache controller. Each cache block is fetched, decrypted, and then sent to the I-cache. To support AES, a separate AES decryption unit is instantiated to decrypt each block.

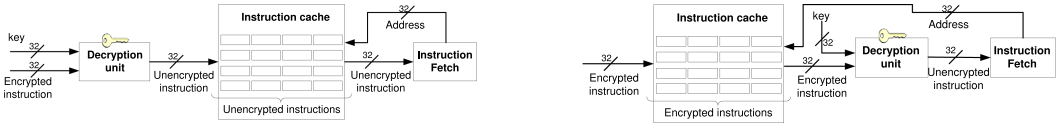
reside decrypted in the I-cache. As shown in Section 5.2, even with AES, the runtime overhead is acceptable due to the locality of the I-cache accesses.

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

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

The 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*.

Figure 3 presents a high level overview for ISR support when using the AES cipher algorithm, and the corresponding MMU that is required. As we can see, every time the I-cache requests an address, the ASIST controller will request the 16 byte aligned memory block (four LS bits of the address equal to zero) from the AHB interface. This is essential, since the I-cache deploys critical word first algorithm, i.e., the address that caused the cache miss will be requested first. The ASIST



(a) Decryption before the I-cache

(b) Decryption after the I-cache

Fig. 4. Alternative choices for the placement of the decryption unit in the ASIST-enabled processor.

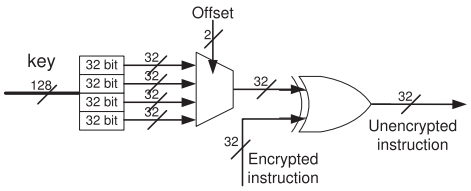


Fig. 5. Decryption using XOR with 128-bit key. Based on the last two bits of the instruction's address (offset) we select the respective 32-bit part of the 128-bit key for decryption.

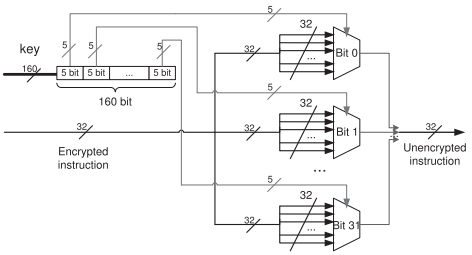


Fig. 6. Decryption using Transposition with 160-bit key. The implementation needs 32 multiplexers with all the 32 bits of the encrypted instruction as input lines in each one.

controller will request and fetch each word of the cache line using the AHB protocol and fill a 128-bit buffer. When the whole block is fetched the buffer is forwarded along with the decryption key to the AES decryption unit. The AES decryption unit we deployed expands the key on each round and requires 12 cycle to decrypt each block. When the block of instructions is decrypted each instruction is forwarded to the I-cache controller in the same manner the vanilla processor's AHB interface would, hence we preserve the critical word first policy. While these extra steps needed on each I-cache miss seem to impose significant performance overhead, our evaluation results indicate that we can amortize a vast percentage of the overhead in most cases due to the I-cache locality.

3.2.3 Decryption Algorithms and Key Size. Figure 5 shows the implementation of XOR decryption with 128-bit key. Since each encrypted instruction in our architecture is a 32-bit word, we need to select the proper 32-bit part of the 128-bit key, the same part that was used in the encryption of this instruction. Thus, we use the two last bits of the instruction's address to select the correct 32-bit part of the 128-bit key using a multiplexer and, finally, decrypt the instruction. The same approach is used for XOR decryption with other key sizes, multiple of 32 bits. The implementation of decryption with Transposition, as shown in Figure 6, requires more hardware. This is because it needs 32 multiplexers, one per bit of the decrypted instruction. Each multiplexer has 32 input lines with all the 32 bits of the encrypted instruction, to choose the proper bit.

Using AES, we have increased area overhead due to the fact that we added an AES decryption unit in the processor. However, modern processors are equiped with cryptography accelerators; thus, we can safely consider it a useful addition. For AES key we utilize the 128 least significant bits of the Transpotition key. Our AES decryption unit, expands the key per-round. It also has five select lines that define the selection of the input bit at each position. The select lines of each multiplexer are part of the 160-bit key. Besides the additional hardware, the runtime operation of Transposition is equally fast with XOR, as it does not spend extra cycles. When using AES,

the runtime overhead is slighly increased due to the complexity of the encryption algorithm, i.e., 12 cycles per 16 byte block instead of on-cycle for XOR and traspotition. To dynamically select the decryption algorithm and key size, we have added a memory mapped register: *asist_mode*.

3.2.4 Return Address Encryption. To transparently protect a system against return-to-libc and ROP attacks [17, 55], 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 [25], which uses `mprotect()` to hide the canary table from attackers. However, we take advantage of the two hardware key registers, which are not accessible by an 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, 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.

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.

We note that return address encryption was first introduced in the initial ASIST implementation [47], as well as in other proposed mechanisms [29, 66], to prevent the disclosure of the text pages location. Encryption in those mechanisms is applied to all code pointers. In our new design we are not only relying on code pointer obfuscation for preventing gadget discovery; by encrypting the text segment with a strong encryption scheme (AES), even the disclosure of code pointers is not sufficient for an attacker to discover usable gadgets. Finally, in the case where an attack can be mounted without disclosing the text segment of the process (e.g., indirect JIT-ROP), any binary obfuscation technique can be used along with our ISR with strong encryption to provide sufficient protection. Such attacks require extensive knowledge of the victim application to succeed [32].

3.3 Operating System Support

In this section, we describe the extra functionality needed in the operating system to support the ASIST hardware features that we describe in Section 3.2.

3.3.1 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 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, then we allocate a new anonymous page that is not backed by any file. Upon the reception of the requested page from disk (or buffer cache), we encrypt its data with process's key and copy it at the same step into the newly allocated page. Then, the new page is mapped into the process's address space.

3.3.2 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 separate code from data into different sections while building the kernel image. The *oskey* register saves the key of kernel's encrypted code. We also modified the bootloader to read and store the kernel's key into the *oskey* register with a *sta* instruction, just before the kernel's execution. Since *oskey* is initialized with zero, which has no effect in XOR decryption that is also default, the unencrypted code of the bootloader can be successfully executed in the randomized processor. In case of AES encryption, if the key is zero, our decryption unit will not decrypt fetched instructions.

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

4 ASIST PROTOTYPE IMPLEMENTATION

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

4.1 Hardware Implementation

We implement ASIST on Leon3 SPARC V8 processor [1], a 32-bit open-source synthesizable processor [28]. Leon3 uses a single-issue, seven-stage pipeline, 8 register windows, and a 16 KB 2-way

Table 2. Additional Hardware Used by ASIST

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

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. When using AES the overhead is slightly over 10%.

set associative D-cache. To study the overhead that will be imposed in a typical processor we also configured our Leon3 with 16-KB (2-way associative), 32-KB, and 64-KB (4-way associative) I-cache sizes, respectively. The cache has a single level and is pure Harvard architecture. Finally, we synthesized and mapped the modified ASIST processors on a Xilinx XUPV5 ML509 FPGA board [67]. The FPGA has 256 MB DDR2 SDRAM memory and operates at 80-MHz clock frequency. The source code of our implementation is available [30].

4.2 Additional Hardware

Table 2 shows the results of the synthesis for three different hardware implementations of ASIST: (i) using XOR decryption with 32-bit and 128-bit keys, respectively; (ii) using decryption with Transposition and a 160-bit key; and (iii) using AES decryption with 128-bit key. We also show the case for the unmodified Leon3 processor as a baseline to measure the additional hardware used by ASIST to implement the 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 for both flip flops (6.62% increase) and lookup tables (6.87% increase). This is due to the larger circuit used for the hardware implementation of Transposition, which consists of 32 multiplexers with 32 input lines each, as we showed in Section 3.2.3. Finally, the AES implementation results in approximately 10% more hardware, mostly due to its larger complexity. Obviously, the extra hardware required can be decreased in cases where the processors already contain cryptographic accelerators (such as the AES-NI [7] contained in the majority of recent Intel and AMD CPUs). Such accelerators can be used directly by our ISR implementation, thus amortizing a large percentage of the area overhead.

4.3 Kernel and Software Modifications

As we describe in Section 3.3, we have modified the Linux kernel and its toolchain to provide a full-featured SPARC workstation. In particular, we ported our Linux kernel modifications in 3.8.0 kernel version. We built a cross compilation tool chain with `gcc` version 4.7.2 and `uClibc` version 0.9.33.2 to cross compile the Linux kernel, libraries, and user-level applications. Thus, all programs running in our system (both vanilla and ASIST), including the vulnerable programs that we use for the security analysis, as well as the benchmarks that we use for the performance evaluation, were cross compiled on another PC. We created a new linker script to separate code and data for both static and dynamic code encryption, and align headers, code, and data into separate pages

in case of dynamic encryption. To implement static encryption, we extended `objcopy` with the `---encrypt-code` flag. The key can be provided by the user or randomly chosen.

4.4 Portability to Other Architectures

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

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

4.4.2 Out-of-order Execution. The Leon3 processor that we use for the implementation of ASIST is a simple seven-stage pipelined processor. As such, there are no options available to synthesize an out-of-order core. Even though this prevents us from evaluating ASIST in terms of performance and circuit area overhead, still we believe that our design can be used in out-of-order execution cores without any modifications. This is due to the fact that ASIST can decrypt every instruction right before being stored in the I-cache, hence not modifying the processor core itself. In terms of performance overhead, we can extrapolate that the resulting overheads will be similar with the in-order execution that are presented in Section 5.2. The overhead of ASIST highly correlates with the I-cache miss rate, rather than the core's architecture. In terms of area overhead, we can expect that our additional hardware will be a lower percentage of the processor than the current, in-order, implementation of ASIST, since out-of-order cores require more area.

5 EXPERIMENTAL EVALUATION

We now present the experimental evaluation of our ASIST prototype in terms of security and performance. As described in Section 4, our prototype is implemented on a FPGA, running Linux kernel v2.6 or v3.8. For the security evaluation, we also configured the networking of our base system and the corresponding Etherner adapter, in order to enable remote exploitation attempts. We also install a `ssh` server to connect to the Linux OS and execute the cross-compiled benchmark programs. The output of each benchark is redirected to local files, thus avoiding any network delays.

5.1 Security Evaluation

5.1.1 Synthetic Attacks. In our first experiment, we use a vanilla v.2.6.21 kernel, which does not properly implement a non-executable stack on SPARC. We build a custom program with a typical

Table 3. Representative Subset of Code Injection Attacks Tested with ASIST

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

We see that ASIST is able to successfully prevent code injection attacks targeting vulnerable user-level programs as well as kernel vulnerabilities.

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

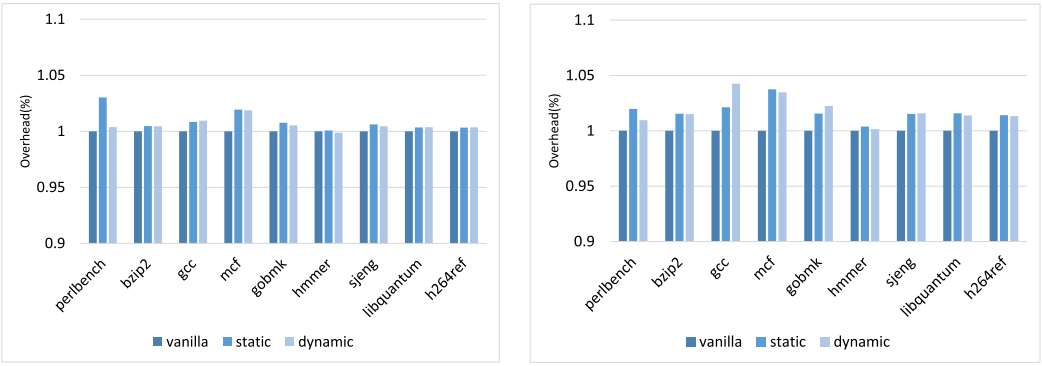
5.1.2 Real Attacks. To demonstrate the effectiveness of ASIST at preventing real code injection attacks that exploit user- or kernel-level vulnerabilities, we test attacks, shown in Table 3. The first six attacks target buffer overflow vulnerabilities on user-level programs, while the last three attacks, a `NULL` pointer dereference and two buffer overflow vulnerabilities in kernel space.

In addition, we performe similar tests with other known vulnerable programs: Ettercap, which is a packet capture tool, MariaDB database, sendmail, Light HTTPd, and Null HTTPd webservers. These programs were cross compiled with our toolchain and encrypted with our extended `objcopy` tool. The injected shellcode is executed successfully only on the vanilla system, while ASIST always prevents the execution of the injected code and results in an illegal instruction exception.

5.2 Performance Evaluation

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

5.2.1 Micro-Benchmarks. In our first benchmark, we run a representing subset of the integer benchmarks (CINT2006) from the SPEC CPU2006 suite [61], which includes several CPU-intensive applications. Figure 7 and Table 4 shows the slowdown of each benchmark when using ASIST



(a) XOR (b) Transposition

Fig. 7. Percentage of overhead of (a) XOR and (b) Transposition when using the SPEC CPU2006 benchmark suite. We see that both ASIST implementations have negligible runtime overhead compared to the vanilla system.

Table 4. Geometric Mean of the Overheads in SPEC2006 Benchmark Suite

Mode	XOR	Transposition	AES (16KB I-cache)	AES (32KB I-cache)	AES (64KB I-cache)
Static	0.9%	1.76%	12.6%	5.57%	1.08%
Dynamic	0.58%	1.87%	15.16%	8.64%	4.82%

When using a typical size for the I-cache the overhead becomes very low due to the spatial locality of the text segment accesses.

with static and dynamic encryption, respectively, compared to the vanilla system. We observe that both XOR and Transposition impose less than 3% slowdown in all benchmarks. This is due to the hardware-based instruction decryption, which does not add any observable delay. Moreover, the modified kernel performs only minor extra tasks, such as reading the key from the executable file (for static encryption) or randomly generating a new key (for dynamic encryption) once per each execution, while adding only one extra instruction before each context switch. We notice a slight deviation from the vanilla execution time only for three of the benchmarks: `gcc`, `sjeng`, and `h264ref`. For these benchmarks, we observe a slowdown of 1–1.2% in static and 1–1.5% in dynamic encryption, which is probably due to the different linking configurations (statically linked versus dynamically linked shared libraries).

One might expect that the dynamic encryption approach would experience a considerable performance overhead due to the extra memory copy and extra work needed to encrypt code pages at each text page fault. However, our results in Figure 7 indicate that dynamic encryption performs equally well with static encryption. Thus, our proposed approach to dynamically encrypt program code at the page fault handler, does not seem to add any extra overhead.

For the AES implementation, we observe in Figure 8 and Table 4 an increased overhead; this is due to the fact that ~12 cycles are added on each I-cache miss. This overhead is further exaggerated when using dynamic encryption. The performance impact when using the default I-cache size (i.e., 16 KB) is quite impractical for some of the benchmarks. When using larger I-cache though, the performance impact of both static and dynamic solutions drops significantly—for a 64 KB I-cache, only `gcc` and `perlbench` exceeded 10% runtime overhead, due to their high I-cache miss rate. Thus, we can safely assume that even when using AES encryption, ISR is a practical solution.

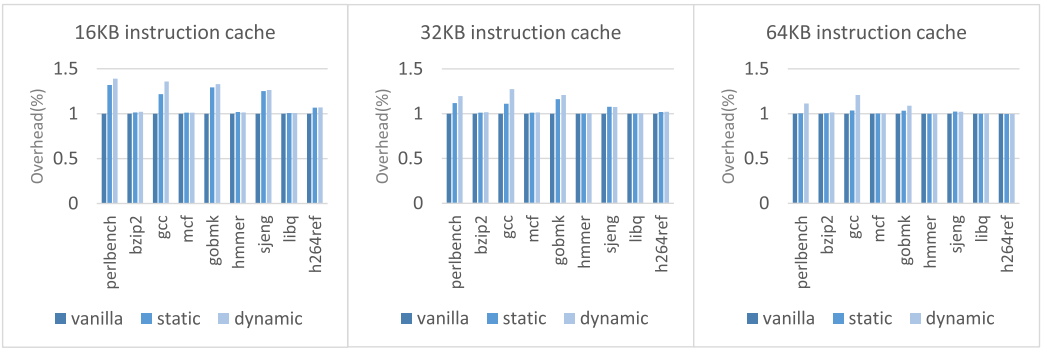


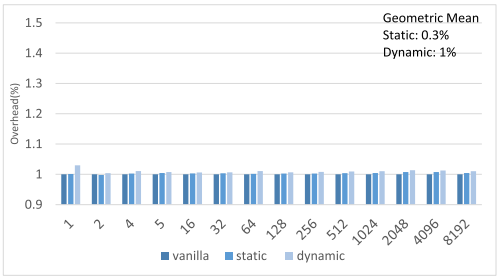
Fig. 8. Percentage of overhead with AES when utilizing different I-cache sizes. We see that ASIST with AES imposes significant overhead when the I-cache size is relatively small and thus the ASIST unit is invoked frequently. When we increase the I-cache size, the runtime overhead is reduced to practical margins. Note that the maximum I-cache size we used in our configurations, is typical for modern commodity processors.

Table 5. Data and Text Page Faults per Second and Percentage during Execution When Running the SPEC CPU2006 Benchmark Suite

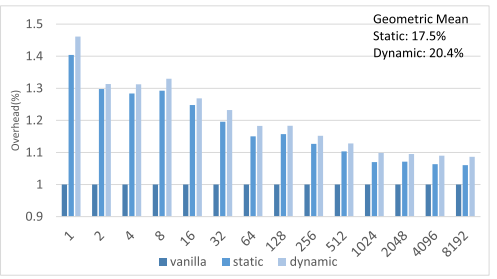
Benchmark	Data page faults rate	Text page faults rate	Data page faults (%)	Text page faults (%)
400.perlbench	38.4964	1.97215	95.1267%	4.87329%
401.bzip2	44.3605	0.193831	99.565%	0.435044%
403.gcc	60.3235	3.93358	93.8784%	6.12164%
429.mcf	51.7769	0.0497679	99.904%	0.0960275%
445.gobmk	25.4735	0.905984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

Text page faults rarely occur, attributing to less than 5% of the total page faults in most benchmarks. This explains the negligible overhead of the dynamic encryption approach.

To better understand the performance of this approach, we further instrumented the Linux kernel to measure the data and text page faults of each process that uses the dynamic encryption mode. Table 5 shows the data and text page faults per second for each benchmark. We see that all benchmarks have a very low rate of text page faults, and most of them experience significantly less than one text page fault per second. Moreover, we observe that the vast majority of page faults are for data pages, while only a small percentage of the total page faults are related to code. The negligible overhead with dynamic code encryption at the page fault handler is due to two main reasons: (i) as we see in Table 5, 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.

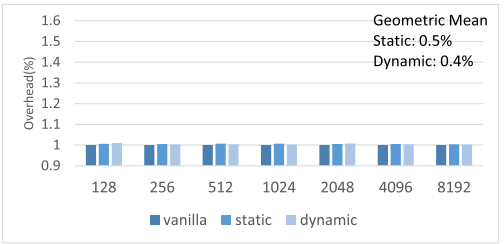


(a) Percentage of overhead when downloading different files from a lighttpd Web server as a function of the file size. We see that ASIST adds less than 1% delay for all file sizes.

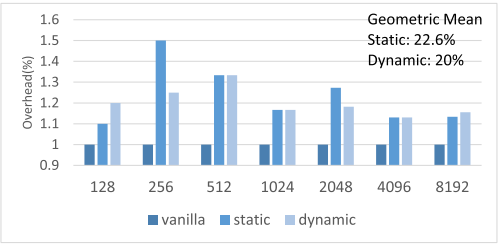


(b) Percentage of overhead when using AES ASIST protected lighttpd downloading different files from a lighttpd Web server as a function of the file size. We can see that for the typical size of HTTP responses the overhead percentage is 9%.

Fig. 9. Percentage of overhead of lighttpd in all ISR modes.



(a) Percentage of overhead 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.



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

Fig. 10. Percentage of overhead of sqlite3 in all ISR modes.

5.2.2 Real-world Applications. First, we run the `lighttpd` web server in a vanilla system and in the two encryption approaches (i.e., static and dynamic of ASIST). Figure 9 shows the slowdown of the average download time for different file sizes. We see that ASIST does not impose any considerable delay, as the download time remains within 1% of the vanilla system for all file sizes. We notice that both static and dynamic encryption implementations perform equally good. We measure the page faults caused by `lighttpd`: 261 data page faults per second, while only 0.013 text page fault per second. Moreover, most of these text page faults occur during the first few milliseconds of the `lighttpd` execution, when the code is loaded into memory. When running `lighttpd` with AES encryption and a 64-KB I-cache (Figure 9(b)), we notice that despite the overhead measured when using small datasets, the overhead is reduced when the HTTP response size exceeds 2 KB. [2]

Next, we run a `sqlite3` database using the vanilla and the three ASIST setups. To evaluate `sqlite3` we implement a benchmark that reads a large text file and updates a SQL table, using the C/C++ SQLite interface. Figure 10(a) shows the slowdown when inserting data into the database as a function of the number of insertions. 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

[2]The typical workload size for HTTP responses is well over 2 KB on average, except from MicroBlogs, where the average size is 1 KB [42].

with enough time to amortize the encryption overhead. The same behaviour is noticed when we run `sqlite3` with AES encryption and a 64-KB I-cache (Figure 10(b)).

6 RELATED WORK

Instruction Set Randomization. ISR was initially introduced as a generic defense against code injections by Kc et al. [37] and Barrantes et al. [8, 9]. To demonstrate ISR, they proposed implementations with `bochs` [40] and `Valgrind` [46], respectively. Hu et al. [35] implemented ISR with Strata SDT tool [52] using AES as a stronger encryption for instruction randomization. Boyd et al. [16] propose a selective ISR to reduce the runtime overhead. Portokalidis and Keromytis [50] implemented ISR using Pin [44] 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 support. Polyglot [56] is similar with ASIST but does not offer dynamic encryption, and hence blind ROP attacks are still possible. Shuffler [66] protects only user-space processes by periodically re-randomizing their address space layout. The re-randomization is perfomed by a separate thread that runs in parallel, one for each process. Morpheus [29] also uses code pointer obfuscation and instruction set randomization.

Defenses against code injection attacks. Modern hardware platforms support non-executable data protection, such as the No eXecute (NX) bit [48]. NX bit prevents data from being executed, so it can protect against code inject attacks without performance degradation. However, its effectiveness depends on its proper use by software. For instance, an application may not set the NX bit on all data segments due to backwards compatibility constraints, self-modifying code and bad programming practices. We believe that ASIST can be used complementary to NX bit, in case that NX bit may not be applicable or can be bypassed. For instance, many ROP exploits use the code of `mprotect()` to make executable pages with injected code, bypassing the NX bit protection. This way, they can execute arbitrary code to implement the attack without the need of more specific gadgets, which may not be easy to find, e.g., due to the use of ASLR. In contrast, these exploits cannot execute injected code in a system using ASIST, as this code will not be correctly encrypted. Thus, ASIST with ASLR provides a stronger defense.

Attacks demonstrated by Snow et al. [58] is also able to bypass NX bit and ASLR using ROP. First, it exploits a memory disclosure to map process's memory layout, and then it uses a disassembler to dynamically discover gadgets that can be used for the ROP attack. 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 user space memory, while the stronger variant, which utilizes the AES encryption algorithm, aims to avoid inferring the key, even when the ciphertext and plaintext are known. SecVisor [53] protects the kernel from code injection attacks using a hypervisor to prevent unauthorized code execution. While SecVisor focuses on kernel's code integrity, ASIST prevents the execution of unauthorized code in both user and kernel level.

Defenses against buffer overflow attacks. StackGuard [25] uses canaries to protect the stack, while PointGuard [24] encrypts all pointers while they reside in memory and decrypts them before they are loaded into a register. Both techniques are implemented with compiler extensions, so they require program recompilation. In contrast, BinArmor [57] protects existing binaries from buffer overflows by discovering the data structures and then rewriting the binary. CFI is a mechanism that aims to limit the locations where a code pointer can point [6]. Many different policies have been proposed to reduce the performance overheads and reduce the analysis required [27, 68]. However, it has been shown that relaxed CFI policies are not sufficient against code-reuse attacks [22].

Other randomization-based defenses. ASLR [49] randomizes the memory layout of a process at runtime or at compile time to protect against code-reuse attacks. Giuffrida et al. [31] propose an approach with address space randomization to protect the operating system kernel. Bhatkar et al. [10] present randomization techniques for the addresses of the stack, heap, dynamic libraries, routines, and static data in an executable. Wartell et al. [64] randomize the instruction addresses at each execution to address code-reuse attacks. Jiang et al. [36] prevent code injections by randomizing the system call numbers.

7 CONCLUSIONS

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

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

REFERENCES

[1] [n.d.]. The SPARC Architecture Manual, Version 8. Retrieved from www.sparc.com/standards/V8.pdf.

[2] [n.d.]. USA National Vulnerability Database. Retrieved from http://web.nvd.nist.gov/view/vuln/statistics.

[3] 2006. Linux Kernel Remote Buffer Overflow Vulnerabilities. Retrieved from http://secwatch.org/advisories/1013445/.

[4] 2007. OpenBSD IPv6 mbuf Remote Kernel Buffer Overflow. Retrieved from http://www.securityfocus.com/archive/1/462728/30/0/threaded.

[5] 2008. Microsoft Windows TCP/IP IGMP MLD Remote Buffer Overflow Vulnerability. Retrieved from http://www.securityfocus.com/bid/27100.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (2009), 4.

[7] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. 2010. Breakthrough AES performance with intel AES new instructions. White Paper, June (2010), 11.

[8] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. 2005. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.* 8, 1 (2005).

[9] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM Conference on Computer and Communications Security*.

[10] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium*.

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

[12] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE, 227–242.

[13] Dion Blazakis. 2010. Interpreter exploitation: Pointer inference and JIT spraying. In *BlackHat DC Conference* (2010).

[14] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*.

[15] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*.

[16] Stephen W. Boyd, Gaurav S. Kc, Michael E. Locasto, Angelos D. Keromytis, and Vassilis Prevelakis. 2010. On the general applicability of instruction-set randomization. *IEEE Trans. Depend. Sec. Comput.* 7, 3 (2010).

[17] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the ACM Conference on Computer and Communications Security*.

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

[19] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys'11)*.

[20] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-control-data attacks are realistic threats. In *Proceedings of the USENIX Security Symposium*.

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

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

[23] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. 2001. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the USENIX Security Symposium*.

[24] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. PointguardTM: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the USENIX Security Symposium*.

[25] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium*.

[26] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2008. Real-world buffer overflow protection for userspace & kernelspace. In *Proceedings of the USENIX Security Symposium*.

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

[28] Gaisler Research. 2005. Leon3 synthesizable processor. Retrieved from http://www.gaisler.com.

[29] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, et al. 2019. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 469–484.

[30] George Christou. 2020. ASIST Leon3 source code. Retrieved from https://github.com/G3org1o/grlib-asistmmu-aes.

[31] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the USENIX Security Symposium*.

[32] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. 2018. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P'18)*. IEEE, 227–242.

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

[34] Joseph L. Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. 2012. A case for unlimited watchpoints. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*.

[35] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. 2006. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'06)*.

[36] Xuxian Jiang, Helen J. Wangz, Dongyan Xu, and Yi-Min Wang. 2007. RandSys: Thwarting code injection attacks with system service interface randomization. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS'07)*.

[37] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[38] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the USENIX Security Symposium*.

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

[40] Kevin P. Lawton. 1996. Bochs: A portable PC emulator for Unix/X. *Linux J.* 1996, 29es (1996), 7-es.

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

[42] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proceedings of the ACM SIGARCH Computer Architecture News*, Vol. 41.

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

[44] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*.

[45] Nergal. 2001. The advanced return-into-lib(c) exploits: PaX case study. *Phrack* 11, 58 (2001).

[46] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*.

[47] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. 2013. ASIST: Architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*.

[48] Linda Dailey Paulson. 2004. New chips stop buffer overflow attacks. *IEEE Comput.* 37, 10 (2004).

[49] PaX Tream. 2001. Homepage of PaX. Retrieved from http://pax.grsecurity.net/.

[50] Georgios Portokalidis and Angelos D. Keromytis. 2010. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'10)*.

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

[52] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. 2003. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'03)*.

[53] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'07)*.

[54] Simha Sethumadhavan, Salvatore J. Stolfo, Angelos Keromytis, Junfeng Yang, and David August. 2011. The SPARCHS project: Hardware support for software security. In *Proceedings of the Systems Security Symposium Workshop (SysSec Workshop'11)*.

[55] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security*.

[56] Kanad Sinha, Vasileios P Kemerlis, and Simha Sethumadhavan. 2017. Reviving instruction set randomization. In *Proceedings of the 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST'12)*. IEEE, 21–28.

[57] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2012. Body armor for binaries: Preventing buffer overflows without recompilation. In *Proceedings of the USENIX Annual Technical Conference (ATC'12)*.

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

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

[60] Ana Nora Sovarel, David Evans, and Nathanael Paul. 2005. Where's the FEEB? The effectiveness of instruction set randomization. In *Proceedings of the USENIX Security Symposium*.

[61] Standard Performance Evaluation Corporation (SPEC). 2006. SPEC CINT2006 Benchmarks. Retrieved from http://www.spec.org/cpu2006/CINT2006.

[62] Nathan Tuck, Brad Calder, and George Varghese. 2004. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'04)*.

[63] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Improving integer security for systems with KINT. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI'12)*.

[64] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[65] Yoav Weiss and Elena Gabriela Barrantes. 2006. Known/chosen key attacks against software instruction set randomization. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'06)*.

[66] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[67] Xilinx. 2011. Xilinx University Program XUPV5-LX110T Development System. Retrieved from http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf.

[68] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security'13)*. 337–352.

Received May 2019; revised May 2020; accepted August 2020