Reviving Instruction Set Randomization

Kanad Sinha
Columbia University
kanad@cs.columbia.edu

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

Simha Sethumadhavan
Columbia University
simha@cs.columbia.edu

***Abstract*—Instruction set randomization (ISR) was proposed early in the last decade as a countermeasure against code injection attacks. However, it is considered to have lost its relevance; with the pervasiveness of code-reuse techniques in modern attacks, code injection no longer remains a foundational component in contemporary exploits.**

This paper revisits the relevance of ISR in the current security landscape. We show that prior ISR schemes are ineffective against code injection, but can be made effective against code-reuse attacks, and even counter state-of-the-art variants, such as "just-in-time" ROP (JIT-ROP). Yet, certain key architectural features are necessary for enabling these capabilities. We implement a new ISR system, namely Polyglot, on a SPARC32-based Leon3 FPGA that runs Linux. We show that our system incurs a low performance overhead (4.6% on a subset of SPEC CINT2006) and defends against real-world (JIT-)ROP exploits, while still supporting critical features like page sharing. Polyglot is also the first ISR implementation to be applicable to the entire software stack: from the bootloader to user applications.

I. Introduction

Instruction set randomization (ISR) was initially proposed as a mitigation against code injection [1], [2]. ISR involves *randomizing* the underlying instruction-set architecture (ISA) of a CPU, giving the impression of a unique instruction set to every program. For instance, the opcode `0xa` may denote the `XOR` instruction on one application, but may be invalid on a different one (that executes on the same platform). This prevents an attacker from using the *same* exploit on multiple targets, as any injected (shell)code must adhere to the (unique) ISA used by the vulnerable program to be effective. ISR implementations typically "emulate" the random ISA using encryption; code is encrypted at the binary level and decrypted in-memory, before execution.

However, ISR has a major drawback that impedes its widespread adoption: it is completely ineffective against code-reuse attacks, which are the cornerstone of modern, real-world exploits [3], [4]. This is because code-reuse attacks, as the term implies, stitch together code pieces that are already present in the address space of a running process. Additionally, once an attacker has the means to overcome techniques like W⊕X, ISR can be trivially bypassed as well. Hence, even as a defense against code injection, ISR is no more effective than the established techniques. In short, the previously-proposed ISR schemes suffer from one, or more, of the following issues:

① **Unfavorable performance–security trade-offs.** Since program instructions are decrypted at runtime, the decryption process falls squarely in the critical path of instruction execution and the associated latency is hard to amortize.

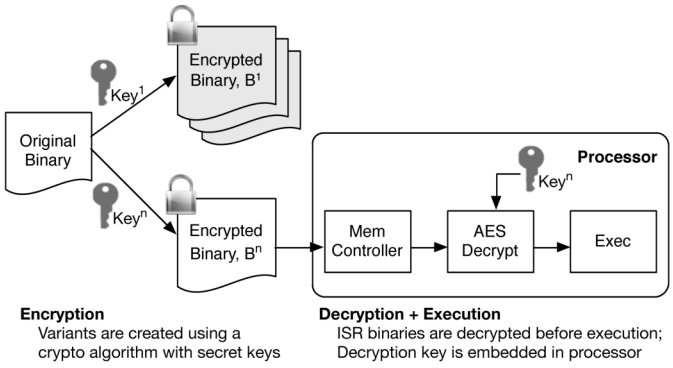

Fig. 1: High-level overview of Polyglot.

② **No self-protection.** Most previous solutions are software-based, they expose a large trusted computing base (TCB), and can be easily turned off, as they execute at the same privilege level with the component(s) they protect.

③ **No support for shared libraries and page sharing.** ISR schemes that lack support for code sharing (inevitably) result in large memory overheads [5].

④ **Archaic threat model.** Previous approaches do not consider memory disclosure vulnerabilities as part of their threat model. However, arbitrary memory-read capabilities are integral to modern attacks and on par with the threat model(s) of recent studies [6].

In this paper we propose *Polyglot*, a hardware-based ISR scheme that concurrently addresses all the above concerns. Polyglot not only improves the traditional security properties of ISR, but also counters state-of-the-art code-reuse attacks, which are a novel and more relevant target. We utilize strong encryption (i.e., AES [7] and ECC [8]), successfully overcoming the challenges of unattainable performance by removing decryption from the critical path (using microarchitectural optimizations); this grants our scheme additional properties such as code secrecy. Unlike prior schemes, we encrypt at the *page*, instead of application, granularity. This not only enables richer diversification, but also allows us to support page sharing and seamlessly apply ISR to system software (OS, hypervisor, *etc.*). We are also the first to demonstrate how ISR can be logically extended to operate from system boot time, when the memory management unit (MMU), and hence paging, is disabled; our implementation of ISR is available from the very first instruction that the system executes.

Furthermore, we keep a small hardware TCB by limiting our modifications to within the processor. Figure 1 provides an overview of Polyglot. Binaries are encrypted page-wise, with AES, while upon execution, the hardware memory controller (with the help of the OS) decrypts the executing instructions as they are transferred into the code cache, so that the program remains encrypted in memory.

II. Background and Motivation

In this section, we give a brief overview of previous work on ISR, and claim that in the face of modern exploitation techniques, ISR is completely ineffective. Subsequently, we motivate a novel way in which ISR can become valuable again, when combined with other protection mechanisms, albeit only if strong encryption is employed.

A. Previous ISR Schemes

Most prior ISR attempts are software based (see Table I), and typically implement randomization using dynamic binary instrumentation tools, such as Pin [13], or platform emulators, like Bochs [14]. Obviously, an attacker can turn off, or subvert, such components as they execute at the same privilege level as the protected application. Apart from other practicality issues, they also exhibit significant slowdowns [11] and were demonstrated to be bypassable [15], [16], due to their use of weak XOR-based encryption.

ASIST [12] sidesteps problems with performance and (the lack of support for) shared libraries by providing hardware support for ISR, and incorporates the best practices of most of the previous techniques. ASIST allows two types of weak encryption: XOR and transposition. The encryption keys are unique to every process, and can be generated either at compile time, in which case the application statically links with all its dependencies, or at load time, where the pages are encrypted dynamically. The latter mode allows shared libraries, but does not allow sharing them between applications, thus incurring a significant (space) overhead [5]. Absence of page sharing also precludes the copy-on-write primitive, significantly increasing the overhead of `fork`.

B. ISR against Code-Reuse Attacks

Code-reuse techniques are the attackers' response to the increased adoption of hardening mechanisms, like address space layout randomization (ASLR) and non-executable memory (NX), by commodity systems. The main idea behind code-reuse is to construct the malicious payload by reusing instructions already present in the address space of a vulnerable process [3]. This powerful technique gives the attacker the same level of flexibility offered by arbitrary code injection, without injecting any new code at all; the malicious payload consists of just a sequence of *gadget* addresses intermixed with any necessary data arguments.

ISR is *fundamentally* ineffective against code-reuse attacks (CRAs), since attackers can construct their payload without knowing how the instructions have been scrambled. One only needs to know the *location* of the appropriate gadgets.

In the presence of various code-randomization schemes [17], state-of-the-art CRAs have been evolved to discover gadgets dynamically, at runtime, by scanning code pages [6]. To prevent this, two conditions have to be met:

① The host binary should differ from the attacker's copy.
② Code should not be readable.

These two conditions are sufficient, as an attacker can neither scan binaries at the host, nor use their own copy in conjunction with the diversified copy to mount a CRA. (Note that this does not rule out data-only attacks [18].)

To this end, we propose combining ISR with a fine-grained (code) randomization scheme to thwart state-of-the-art CRAs. While the latter satisfies condition ① an ideal ISR implementation can also *indirectly* prevent condition ② by revealing (ideally irreversible) encrypted text when code is read.

However, if ISR, plus diversification, is all that is needed, can we just combine code randomization with any of the previous ISR proposals? In other words, can we use weak encryption to fulfill the above conditions? Unfortunately, encryption schemes such as XOR and bit transposition can be easily bypassed, even under the presence of fine-grained diversification; during our preliminary experiments we were able to leak keys for the two schemes easily, using entirely architecture- and ABI-independent methods (due to the lack of space we will refrain from elaborating more on the subject). Hence, we argue that using stronger encryption, at low cost, is *pivotal* in providing ISR-based protection.

III. System Architecture

In this section, we present Polyglot's architectural design, detailing our software and hardware modifications, and how they inter-operate to achieve our goals.

A. Software

Binary Generation. To create an "ISRized" binary, we symmetrically encrypt a diversified version of it, at page granularity, with randomly generated keys. (Note that only executable sections are encrypted.) These key-to-address mappings are then asymmetrically encrypted using the target processor's public key and packaged into the binary itself. Since code is encrypted at a page granularity, the executable, and its required shared libraries, possibly encrypted by different sources, are able to interoperate. Lastly, asymmetric encryption ties the binaries to their respective hosts.

Binary Execution. The dynamic loader and the OS are responsible for extracting the encrypted keys from ISRized binaries. In particular, the OS is in charge of them, during its execution lifetime, and for setting up the process' page tables, as well as its own, in a format expected by the hardware. (Note that our scheme allows code pages to exist in plaintext if necessary.) Additionally, since we encrypt at page-granularity, code sharing for shared libraries, as well as for forked processes, is readily supported—i.e., by using the same translation entry among the page tables of processes that share a particular page (see Figure 2).

Proposal	SW/HW Based	Encryption	Granularity	Shared Libraries*	Code Sharing	Self-modifying Code	Scope	Perf. Overhead
Barrantes et al. [2]	SW	XOR	Proc.	✗	✗	✗	App.	High
Kc et al. [1]	SW	XOR	Proc.	✗	✗	✗	App.	High
Hu et al. [9]	SW	AES	Proc.	✗	✗	✗	App.	High
Boyd et al. [10]	SW	XOR	Proc.	✗	✗	✗	App.	Med.
Portokalidis et al. [11]	SW	XOR	Proc.	✓	✓	✓	App.	Med.
Papadogiannakis et al. [12]	HW	XOR, Trans/on	Proc., Kernel	✓	✗	✓	App., Kernel	Negl.
Polyglot	HW	AES, ECC	Mem. page	✓	✓	✓	App., Kernel, Bootloader	Low

TABLE I: Comparison of various ISR proposals. (*Shared library support does not necessarily imply sharing them across processes, unless code sharing is allowed.)

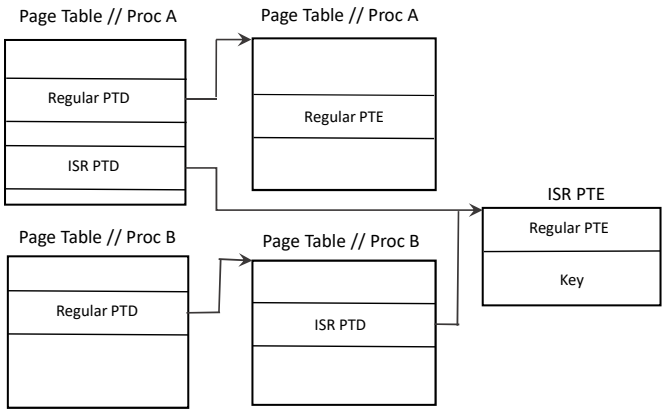

Fig. 2: Page table modifications. The ISR PTE corresponds to a page shared between processes A and B. PTD indicates a page table descriptor, which is a pointer to the next level, whereas PTE is the last-level translation.

Supporting ISR at the kernel level is achieved simply by changing the kernel's own page mapping(s) to an ISR-PTE version. Overall, our modifications added ∼1100 LOC to the Linux kernel (v3.8.1). Since paging is disabled at system boot, we randomize the bootloader by encrypting the whole image according to its layout in physical memory, so that encrypted execution is enabled from the very first instruction. Care is taken, however, to ensure that when the MMU is turned on, and paging enabled, the respective keys match (i.e., before and after enabling virtual addressing).

B. Hardware

Limiting the TCB to only the processor imposes numerous challenges on our design. While previous work supports only weakly-encrypted code, to avoid performance overheads, our goal is particularly ambitious since we not only perform symmetric AES decryption of code, but also asymmetric ECC decryption of metadata (the latter operation can be orders of magnitude slower than the former). Furthermore, since our TCB is just the processor, we want code and keys to be encrypted, at all times, while outside the chip. We achieve this by modifying the instruction page fault and instruction cache miss pathway as shown in Figure 4.

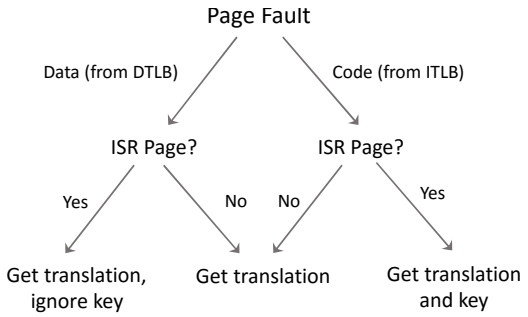

Fig. 3: ISR page fault handling flowchart.

1) Instruction Page Fault: To accommodate per-page encryption, we introduce a new type of page table entry (PTE) for randomized (i.e., ISR-encrypted) pages. Specifically, we require that the corresponding PTEs contain the actual translation followed by the key for the respective page, as shown in Figure 2. However, since ISR-PTEs become longer than a word, and do not conform to a power-of-two alignment, they break the conventional PTE fetching mechanism. We solve this by extending the hardware page walk by another level—i.e., the penultimate level, which is part of the page tables, contains the physical address of the actual entry. The hardware page walk scheme is, hence, agnostic to how they are arranged in memory (contiguous or discrete).

On a page fault, the origin of the fault (DTLB or ITLB) determines whether a code or data page is expected (see Figure 3). If the code faults, the table walk proceeds as usual until an entry with the ISR type is encountered. The walk mechanism procures encrypted entry, decrypts it to obtain the page key and translation, which are then deposited into a modified ITLB (step ② in Figure 4). We add the requisite ECC-163 and SHA-256 accelerators to the MMU to carry out the decryption according to the Elliptic Curve Integrated Encrypted Scheme (ECIES) [8].

On the other hand, if an ISR page is encountered on a data page walk, then the key is ignored and the respective translation is delivered (as is) to the DTLB. This effectively allows data and code to exist inside the same page: code accesses to a randomized page use decryption, while data accesses to the same page fetch contents as is. This means that

code can be treated as data (i.e., it can be read and written). However, to allow for correct decryption, data and code must either be aligned at cache block width, or, if they exist within the same line, the data must be immutable.

2) Instruction Cache Miss: On an I-cache miss, as instructions are fetched from memory, they are decrypted using the page's key and stored, in plaintext, in the I-cache (Figure 4, step ④). Henceforth, as long as an instruction is not evicted, execution uses its decrypted form. D-cache miss handling remains unaltered. The challenge, however, is that although symmetric decryption is much faster than asymmetric decryption, instruction fetches are more common. This is the main reason prior work opts for inexpensive encryption like XOR.

To overcome this, we use symmetric encryption in counter mode [19]. In this mode, the actual decryption is performed on a counter value, which is then XOR'ed with the actual data to obtain the plaintext. We derive the counter from the lower bits of memory addresses. Notably, since the counter can be determined from the address itself, decryption and fetch can commence *simultaneously.*

The Leon3 SPARC32 implementation has only split L1 caches. Hence, we place the AES decryption engine at the I-cache/memory interface. For modern systems, with multiple cache levels, we propose the decryption engine to be placed at the cache/memory interface for energy and performance reasons. Most importantly, since in such systems the memory latency is much higher than the symmetric decryption latency, the decryption cost can be masked *completely*.

This approach, however, opens the door to the following attack. Assuming a shared L2 cache, suppose that a L2 block is fetched as a response to a data (load) request. If the data is also requested by the L1 I-cache, the block, which came in as data without going through decryption, will be fed as is to the L1 I-cache, thus completely bypassing ISR. The converse case, on the other hand, would allow reading decrypted code as data. We prevent this by tracking instruction and data blocks in all shared caches, by adding a bit to each cache block indicating whether it is instruction or data. This bit is set by tracking the source of the miss, i.e., the instruction or data fetch. Cross-sharing between the split caches is then forbidden, forcing line flushes prior to cross accesses even in the shared levels.

3) MMU-less Execution: Enabling ISR when paging is disabled, during bootstrap, follows the same process except that the key is acquired not from a PTE, but directly from a specific memory location. Design-wise, during this phase, the I-cache changes still remain active, while the page walk modifications are disabled.

C. Design Choice Implications

Encryption Algorithms. We use ECC, instead of RSA (i.e., the more popular asymmetric cipher), since ECC has shorter key lengths and encryption/decryption latencies. Furthermore, the use of counter mode block encryption guarantees that the encrypted code is position dependent and prevents splicing attacks (i.e., copying encrypted code and reusing it elsewhere).

Page Table. Page-level (encryption) granularity implies that brute-forcing, or careful dictionary-like attacks on a particular page, reveals nothing about the rest of the system.

Allowing Data Accesses to ISR Pages. Previously-proposed systems that seek to disallow runtime code-scanning make code pages execute-only. We could easily achieve the same by faulting whenever data page walks encounter an ISR PTE. However, this requires strict segregation between code and data at the page granularity. Although a more secure option, we considered this approach limiting in terms of convenience of development, deployability, and practicality.

Syscall Interface. Our `mmap` variant is used by user applications to map encrypted pages into a process. This is a weak link as it can be exploited by an attacker. Additionally, the original `mmap` is still allowed to load unencrypted code. While the former allowance was indispensable from a practicality standpoint, the latter was necessary for the sake of convenience; variants of our architecture can forbid it. Note that introducing this system call does not make Polyglot more vulnerable to the attack against ASIST that we outlined earlier.

Key Handling. This is one of the most crucial aspects of any crypto-system design. In Polyglot, the symmetric keys are included in the binary. Since these keys are asymmetrically encrypted, Polyglot is safe even against binary code leaks, when an attacker obtains the binary itself or the OS is adversarial[1]. Another implicit assumption is that the private key, specific to a chip, is irretrievable by the attacker. In extreme cases, where the physical tampering of the chip is a concern, the private key could be based on a physically unclonable function [21] that is automatically destroyed if anyone tries to tamper with it.

IV. Security Analysis

A. Effectiveness

Polyglot is basically meant to demonstrate that ISR is not just a defense against code-injection, but can be an effective countermeasure to code-reuse attacks as well. We discussed its effectiveness in the former avatar in Section II. Here we analyze its effectiveness against the latter. Since we rely on static code diversification, we are limited by its robustness; nevertheless, Polyglot is independent of it. We also assume that the encryption used (e.g., AES, ECC) is strong enough to be practically irreversible.

Given the above, static ROP is unattainable, as the code memory differs from the attacker's expectation of it. Furthermore, attempts for directly reading code memory, at runtime, will also fail since the code is encrypted. Thus, Polyglot is secure against direct (JIT-)ROP attacks.

However, since ISR does not modify the data layout, it is susceptible to information disclosures through data. Consequently, if the attacker were to leak function pointers, it is possible to carry out whole-function reuse attacks, such as ret2libc [22] and COOP [23], [24]. This is because although

[1]Note that an adversarial OS can mount additional types of attacks, such as Iago attacks [20].

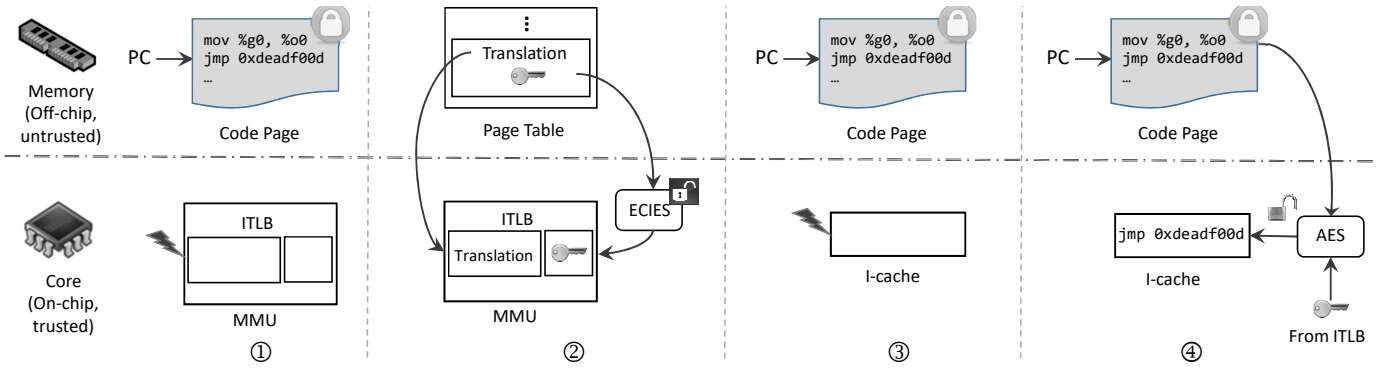

Fig. 4: Hardware decryption in Polyglot. On page faults (left half), ISR PTE contents are brought into the ITLB. On a cache miss (right half), a cache line is decrypted using the page key before dropping into I-cache.

we randomize the structure of functions, we do not change their functionally. Data-space randomization [25], [26] and virtual-table protection mechanisms [27], [28] may be effective in this context. Note, however, that we do prevent some of the recently-published, brute-force memory disclosure attacks. Specifically, we prevent the attack proposed by Seibert et al. [29], simply because our code is encrypted. Moreover, we prevent BROP [30],since an attacker cannot disassemble our code even when they leak the entire `.text` section.

B. Proof-of-Concept Exploit

The main benefit of ISR against CRAs is that gadget-building attempts that rely on (arbitrary) memory disclosure capabilities, for reading a process' code, at runtime, will fail. To assess the effectiveness of Polyglot against direct ROP/JOP attacks, we retrofitted CVE-2013-6282 to Linux kernel v3.8.1. Next, we ported the respective exploit [31] to the SPARC architecture (the original exploit did not use ROP; it used the return-to-user technique [**?**], which relies on forcing the kernel to execute shellcode placed in user space), and verified that it works as expected on the vanilla kernel. We tested the exploit when the same kernel is statically randomized (using a simple scheme that entails function permutation [32] and NOP insertion [**?**]), and, as expected, it failed, as the respective ROP payload relied on pre-computed gadget addresses, none of which remained correct.

As there are no publicly-available JIT-ROP exploits for the SPARC architecture, we retrofitted an arbitrary read-and-write vulnerability in the `debugfs` pseudo-filesystem [33], reachable by user mode. Next, we modified the previous exploit to abuse this vulnerability and disclose the locations of required gadgets by reading the (randomized) kernel `.text` section. Armed with that information, the payload of the previously-failing exploit is adjusted accordingly. We first tested with ISR disabled, to verify that JIT-ROP works as expected, and indeed bypasses the static randomization scheme(s). Then, we enabled ISR and tried the modified exploit again. This attempt failed, as the code could not be read. We also verified encrypted memory contents using a hardware debugger.

V. Evaluation

To evaluate Polyglot, we implemented our design on a Xilinx Virtex5-based XUPV5-LX110T FPGA board. Our implementation is based on the SPARC32-based Leon3 package, and our setup has 256MB of RAM, a portion of which is used as a RAM disk (`ramfs`). On the software side, we used Linux v3.8.1 and uClibc v0.9.33.2. The core utilities were provided through BusyBox v1.23.2. Our system ran with encrypted versions of all the above modules, as well as an encrypted bootloader. For hardware, we used the default Leon3 configuration, sporting an in-order SPARC32 core with no speculation or branch-prediction. Lastly, we use a 64-entry ITLB and a 4-way 32kB I-cache.

It should be noted that even though we run regular workloads on our prototype, the FPGA platform's properties differ from a regular computer in ways that affect our results adversely. Given that TLB and cache misses are the main sources of overhead, overhead reductions are bound to be significant, if structures comparable to those found in contemporary systems are used.[2] Additionally, decryption latency in our case is larger than the latency of memory fetches, while this is not the case for regular systems—our prototype's AES implementation takes 22 cycles to decrypt, while memory fetch, in modern computers, takes about an order of magnitude longer [35].

A. Performance

SPEC. We ran the integer benchmarks of the SPEC CPU2006 benchmark suite [36]. Due to memory limitations on the board, we could only run test inputs and were unable to run all the benchmarks. Note that SPEC benchmarks have been shown to be redundant in metrics (i.e., I-cache and ITLB misses) that are exactly relevant to us [37], and to the extent of our experiments, our results corroborated those findings. Accordingly, `perl` should predictably have similar results as `go`, while `astar` should be similar to `sjeng`. Hence, of the SPECint programs, only `xalanc`'s behavior remains unknown.

[2]For example, ARM Cortex A-15 typically has multi-level caches, and corresponding TLBs, for each level—i.e., a split 32-entry, fully associative L1, and a 512-entry, 4-way unified L2 [34].

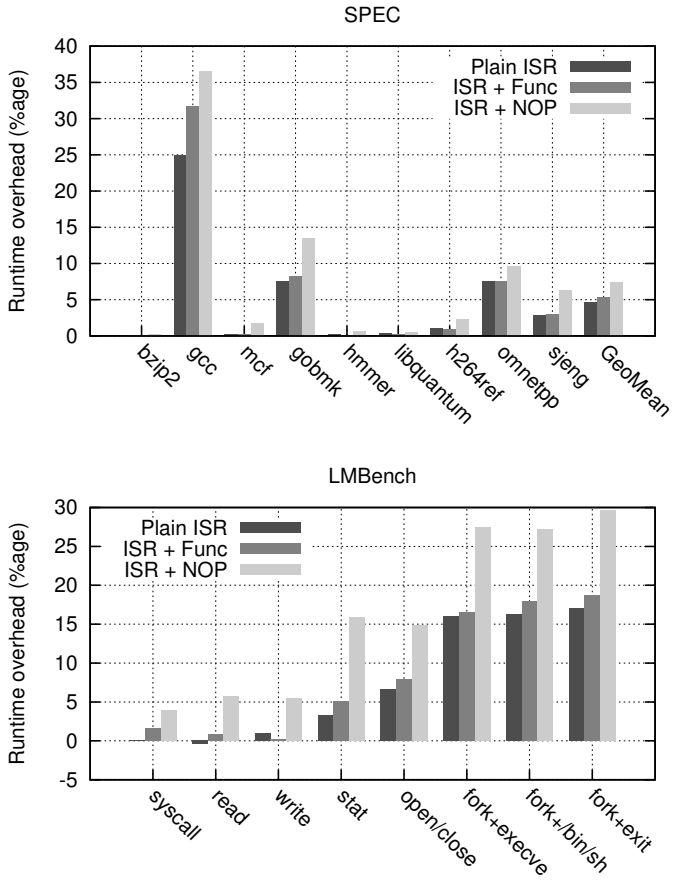

Fig. 5: Performance overhead for SPEC and LMBench.

Results are presented in Figure 5-a. We evaluate Polyglot with three configurations: without randomization, and with function permutation and `NOP` insertion; in the former, the order of functions in the final binary is randomized, while in the latter, we insert up to 8 `NOP`s at function entries and after every call site, preceded by a jump to bypass the `NOP`-sled. Given that we are sensitive to code-misses, these choices are significant (the former does not add to the code size, while the latter does). From the observed data, we see that ISR incurs an overhead of 4.6%. `gcc` performs particularly badly with an overhead of 24.9%, which was a result of an inordinately high rate of ITLB misses (7,376.15 /sec, versus 115.34 /sec for the rest). In fact, if we neglect `gcc`, the rest of the benchmarks have a mean overhead of 2.38%.

For the randomization schemes, function permutation does better, with an overhead of 5.3%, while `NOP` insertion is more expensive: 7.4%. Again, `gcc` was the only outlier with 6% increase in overhead, suffering 15,450 ITLB misses/sec. A similar trend is seen with the `NOP` insertion scheme. Note though that the `NOP`-randomized binaries themselves exhibit a mean overhead of 1.8%. Based on the above observations, we recommend using in-place code-randomization techniques [4] in Polyglot that do not substantially add to code size. With better code-caching support, however, this might be moot.

Kernel. To evaluate the slowdown caused by the encryption of the kernel, we run the LMBench kernel test suite [38] with the same set of variants. We measure the `null` system call as well as a few other (critical) ones: `read`, `write`, `stat`, `open/close`. Importantly, we also measure process creation latency with `fork+{execve, /bin/sh, exit}`. Figure 5-b shows the overhead of encrypted (ISR) over native. We notice similar trends, with overheads for plain ISR being the lowest (0-16%), and ISR+`NOP` exhibiting the highest cost (4-30%).

B. FPGA Implementation Results

Our modifications to the base Leon3 implementation increased LUT usage from 13,986 to 49,724, a significant portion of which was taken up by the cryptoblocks (approx. 17k LUTs) and the ITLB key storage table (approx. 12k LUTs). Since we did not optimize the accelerators for this particular design, we believe that there is plenty of space for improvement (both in terms of area and performance). For instance, instead of using two separate AES-128 accelerators to decrypt a 256B cache-block, we could merge them to a single accelerator, since they share the same key and have almost identical counters. Furthermore, our modifications to the Leon3 distribution synthesizes at the same clock frequency.

VI. Related Work

We covered prior ISR work in Section II. In this section, we survey other hardware- and software-based protection schemes, relevant to Polyglot.

• **Code Diversification.** This line of work seeks to prevent CRAs through diversification. More specifically, this flavor of defenses randomize each instance of a binary, or execution, so that the attacker has only a probabilistic chance of succeeding in finding the necessary gadgets. ASLR [39] and many finer-granularity variants of it [17] were proposed towards this end. The common weakness in this class of defenses is that the randomization is static, and relies on the fact that information about a particular instance (of it) cannot be leaked. It has, however, been shown that attacks based on memory disclosure [6], [23], [29], [30] can dynamically harvest gadgets, thereby disproving this assumption.

Recent defenses against the above can be mainly divided into three categories:

① **Execute-only Memory**. Works in this area [40]–[43] prevent dynamic code memory scanning by making code pages execute-only. Just this measure, however, is not good enough since code pointers can be harvested from data pages as well [44], [45]. Besides, this class of defenses does not support intermingling data and code.

② **Code-pointer Hiding**. Memory disclosures can be mitigated by preventing the leakage of code pointers, direct (e.g., branch targets) or indirect (i.e., function pointers, return addresses), in the first place. Previous work in this category [5], [24], [42], [46] achieved this via a level of hidden or monitored indirection and/or encoding.

③ **Gadget Invalidation**. Such schemes dynamically modify the program's structure (actively or reactively) so that by the time the (harvested) gadgets are employed they are no longer available [44], [47], [48].

Polyglot broadly falls into category ① . While previous proposals actively disallowed reading code, essentially enforcing a no-read property on code pages, we allow code reads while obfuscating readable code. This, in turn, allows us to intermingle code and data, unlike other proposals. We also show our work to be seamlessly applicable to higher-privileged system software. Lastly, none of the previous schemes offers the degree of protection against static binary leaks that we do.

• **Isolated Execution.** These technologies provide a secure, opaque compartment for programs to execute, without the risk of being spied on by other entities, even those executing at a higher privilege level. First introduced in XOM [49], a slew of software [50], [51] and hardware [52]–[54] based techniques have since been proposed; Intel's SGX [55] is an example of the latter category.

In particular, the latter category seeks to provide integrity and confidentiality of both code and data, even in the presence of a malicious operating system. The idea is to achieve security by encrypting code and data outside the TCB (typically the processor), while providing isolation within. Some schemes include memory within their TCB, and simply decrypt the sensitive code at load time, while guaranteeing its integrity thereafter. Others decrypt instructions as they stream into the processor. Although used in other contexts [56]–[58], as far as we know, we are the first to employ symmetric encryption in counter mode in order to mask the instruction-decryption overhead. Additionally, isolation techniques cannot cleanly support shared libraries, due to their strict threat model, requiring extensive changes to software. Design changes further need to ensure the proper modularization of secure components lest the attacker gains entry into a compartment. In brief, we avoid the complexities of the larger problem isolation targets, and, thus, are able to provide a more lightweight solution.

VII. Conclusion

In this paper, we present the design of Polyglot, a hardware-based ISR scheme, which eschews weak cryptography (used by previous ISR proposals) by employing AES and ECC at the (memory) page granularity. We also developed micro-architectural optimizations to reduce performance overheads typically associated with hardware implementations of these cryptographic algorithms. Our solution enables page sharing between applications and strong encryption with low performance overheads. Furthermore, we allow instructions to be encrypted right from system boot. Most importantly, we show how Polyglot can counter state-of-the-art ROP attacks, which ISR was traditionally considered ineffectual against. These features have not been achieved in any prior ISR implementation, and, therefore, provide a promising primitive.

Acknowledgment

We thank the anonymous reviewers and our colleagues, and other members, of the Computer Architecture and Security Technologies Lab (CASTL) at Columbia University, especially Vasilis Pappas, Chester Rebeiro, and Angelos D. Keromytis, for their valuable feedback. This work was supported by grants FA 87501020253 (DARPA), CCF/TC 1054844 (NSF), and N00014-15-1-2173 (ONR), a gift from Bloomberg, and the Alfred P. Sloan Foundation.

References

[1] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks With Instruction-Set Randomization," in *Proc. of CCS*, pp. 272–280, 2003.

[2] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *Proc. of CCS*, pp. 281–289, 2003.

[3] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proc. of ACM CCS*, pp. 552–61, 2007.

[4] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *Proc. of IEEE S&P*, pp. 601–615, 2012.

[5] M. Backes and S. Nürnberger, "Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing," in *Proc. of USENIX SEC*, pp. 433–447, 2014.

[6] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *Proc. of IEEE S&P*, pp. 574–588, 2013.

[7] "FIBS 197, Advanced Encryption Standard (AES)," tech. rep., National Institute of Standards and Technology, 2001.

[8] V. G. Martínez, F. H. Álvarez, L. H. Encinas, and C. S. Ávila, "A Comparison of the Standardized Versions of ECIES," in *Proc. of IEEE IAS*, pp. 1–4, 2010.

[9] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill, "Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation," in *Proc. of VEE*, pp. 2–12, 2006.

[10] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis, "On the General Applicability of Instruction-Set Randomization," *IEEE Trans. Dependable Sec. Comput.*, vol. 7, pp. 255–270, October 2010.

[11] G. Portokalidis and A. D. Keromytis, "Fast and Practical Instruction-Set Randomization for Commodity Systems," in *Proc. of ACSAC*, pp. 41–48, 2010.

[12] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis, "ASIST: Architectural Support for Instruction Set Randomization," in *Proc. of ACM CCS*, pp. 981–992, 2013.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proc. of ACM PLDI*, pp. 190–200, 2005.

[14] K. P. Lawton, "Bochs: A Portable PC Emulator for Unix/X," *Linux Journal*, vol. 1996, no. 29es, 1996.

[15] A. N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? The Effectiveness of Instruction Set Randomization," in *Proc. of USENIX SEC*, pp. 145–160, 2005.

[16] Y. Weiss and E. G. Barrantes, "Known/Chosen Key Attacks against Software Instruction Set Randomization," in *Proc. of ACSAC*, pp. 349–360, 2006.

[17] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated Software Diversity," in *Proc. of IEEE S&P*, pp. 276–291, 2014.

[18] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," in *Proc. of USENIX Sec*, pp. 177–192, 2005.

[19] M. Dworkin, "Recommendations for Block Cipher Modes of Operation: Methods and Techniques," tech. rep., National Institute of Standards and Technology, 2001.

[20] S. Checkoway and H. Shacham, "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface," in *Proc. of ACM ASPLOS*, pp. 253–264, 2013.

[21] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon Physical Random Functions," in *Proc. of ACM CCS*, pp. 148–160, 2002.

[22] S. Designer, "Getting around non-executable stack (and fix)." http://seclists.org/bugtraq/1997/Aug/63, August 1997.

[23] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proc. of IEEE S&P*, pp. 745–762, 2015.

[24] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a TRaP: Table Randomization and Protection Against Function-Reuse Attacks," in *Proc. of ACM CCS*, pp. 243–255, 2015.

[25] S. Bhatkar and R. Sekar, "Data Space Randomization," in *Proc. of DIMVA*, pp. 1–22, 2008.

[26] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu, "A Practical Approach for Adaptive Data Structure Layout Randomization," in *Proc. of ESORICS*, pp. 69–89, 2015.

[27] R. G. K. Dimitar Bounov and S. Lerner, "Protecting C++ Dynamic Dispatch Through VTable Interleaving," in *Proc. of NDSS*, 2016.

[28] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "VTrust: Regaining Trust on Virtual Calls," in *Proc. of NDSS*, 2016.

[29] J. Seibert, H. Okhravi, and E. Söderström, "Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code," in *Proc. of CCS*, pp. 54–65, 2014.

[30] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *Proc. of IEEE S&P*, pp. 227–242, 2014.

[31] Exploit Database, "EBD-31574," February 2014.

[32] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," in *Proc. of USENIX Sec*, pp. 255–270, 2005.

[33] J. Corbet, "An updated guide to `debugfs`." https://lwn.net/Articles/334546/, May 2009.

[34] "Cortex-A15 Technical Reference Manual," tech. rep., ARM, 2011.

[35] D. Levinthal, "Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Core™ 5500 processors," tech. rep., Intel Corporation.

[36] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, September 2006.

[37] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite," in *Proc. of ISCA*, pp. 412–423, 2007.

[38] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in *Proc. of USENIX ATC*, pp. 279–294, 1996.

[39] PaX Team, "address space layout randomization." https://pax.grsecurity.net/docs/aslr.txt, March 2003.

[40] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code," in *Proc. of ACM CCS*, pp. 1342–1353, 2014.

[41] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities," in *Proc. of ACM CODASPY*, pp. 325–336, 2015.

[42] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical Code Randomization Resilient to Memory Disclosure," in *Proc. of IEEE S&P*, pp. 763–780, 2015.

[43] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-Resilient Layout Randomization for Mobile Devices," in *Proc. of NDSS*, 2016.

[44] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming," in *Proc. of NDSS*, 2015.

[45] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi, "Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks," in *Proc. of ACM CCS*, pp. 952–963, 2015.

[46] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks," in *Proc. of ACM CCS*, pp. 280–291, 2015.

[47] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads," in *Proc. of ACM CCS*, pp. 256–267, 2015.

[48] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely Rerandomization for Mitigating Memory Disclosures," in *Proc. of ACM CCS*, pp. 268–279, 2015.

[49] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *Proc. of ACM ASPLOS*, pp. 168–177, 2000.

[50] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," in *Proc. of EuroSys*, pp. 315–328, 2008.

[51] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," in *Proc. of IEEE S&P*, pp. 143–158, 2010.

[52] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing," in *Proc. of ACM ICS*, pp. 160–171, 2003.

[53] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for Protecting Critical Secrets in Microprocessors," in *Proc. of ISCA*, pp. 2–13, 2005.

[54] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution," in *Proc. of IEEE MICRO*, pp. 190–202, 2014.

[55] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proc. of HASP*, 2013.

[56] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," in *Proc. of IEEE MICRO*, pp. 339–350, 2003.

[57] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High Efficiency Counter Mode Security Architecture via Prediction and Precomputation," in *Proc. of ISCA*, pp. 12–24, 2005.

[58] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving Cost, Performance, and Security of Memory Encryption and Authentication," in *Proc. of ISCA*, pp. 179–190, 2006.