Instruction Set Randomization: An Updated Implementation

Jeffrey Kennan
Brown University

Vasileios P. Kemerlis
Brown University

Abstract

Instruction set randomization (ISR), originally designed to prevent code-injection attacks, has been largely overlooked as a defense against modern code hijacking attacks. The idea behind ISR is simple: permute the instruction set a system understands, in order to prevent attempts to inject or read meaningful code. While ISR is effective, in practice it has been passed over in favor of more efficient software- and hardware-based defenses such as non-executable memory and ASLR.

In this work we revisit ISR as a defense mechanism. We design and implement an ISR tool based on dynamic binary instrumentation which supports modern commodity systems, and verify its security guarantees. We also thoroughly test and benchmark our implementation using the GNU coreutils suite. While the performance penalties associated with the use of a dynamic binary instrumentation tool are not insubstantial, the overhead of our defense mechanism on top of that imposed by the instrumentation framework is a mere 3.42%.

1 Introduction

It has long been known that programs written in memory-unsafe languages, such as C and C++, are susceptible to code hijacking attacks. Originally, such attacks came in the form of shellcode injection into the process address space, along with return address overwriting to redirect execution to the injected code. However, the introduction of non-executable memory and subsequent enforcement of $W \oplus X$ memory policies have largely neutralized this attack vector on modern architectures. Thus, code hijacking attacks have moved towards ret2libc and return-oriented-programming (ROP) style atacks, where existing code in the address space of the application is reused for malicious purposes [18].

Address space layout randomization (ASLR) proposes a defense against ROP attacks by shifting the starting addresses of a process's text and data segments each time it is run. The additional entropy introduced by this randomization forces attackers to either brute-force or guess code addresses, and incorrect attempts will likely crash the program and trigger an alarm mechanism [16]. Unfortunately, ASLR fails when a de-randomization attack leaks even a single code address [19]. A more robust, fine-grained variant of ASLR which randomizes code *layout* in addition to its *location*, aims to further increase the barrier to a successful attack; it, too, fails to defend programs that contain a single memory disclosure vulnerability [21]. "Just-in-time ROP" (JIT-ROP) attacks make use of these vulnerabilities to construct exploits at runtime by repeatedly reading code memory to discover "gadgets": short byte sequences that correspond to valid assembly instructions and which can be chained together maliciously.

Instruction set randomization, despite its original application towards preventing code injection, offers a promising new vector against JIT-ROP [13]. ISR creates process-specific randomized instruction sets and trusts the kernel or an instrumentation framework with the derandomization key; instructions are usually derandomized at runtime. In this paper, we introduce a new implementation of ISR for 64-bit Linux using dynamic binary instrumentation (DBI), based on the work of Portokalidis et al. [17]. Our toolchain encrypts binaries ahead of time with a 16-bit XOR key and decrypts instructions at fetch time using Intel's Pin framework. We also present a significantly expanded test suite for performance and correctness, to augment the evaluation presented in that paper.

The rest of this paper is organized as follows: we provide background in § 2, describe our implementation in § 3, present an evaluation of our solution in § 4, and conclude in § 6.

2 Background and Related Work

In this section we introduce ISR and XOR encryption schemes, discuss ISR's ability to prevent both code injection and JIT-ROP attacks, and introduce the threat model on which we based our implementation.

2.1 ISR and XOR Encryption

ISR hardens binaries by creating an "execution environment" unique to the running process [8]. A reversible code transformation changes the "language" which the system understands, and any subsequent attempts to inject shellcode or read memory will do so in an unsupported "language." While primary use cases for ISR focus on randomizing the x86 instruction set, they are by no means limited to it; Boyd et al. [8] have shown its effectiveness against Perl and SQL injection. Notably, neither Perl nor SQL injection require memory corruption vulnerabilities. They can be done on standard web pages and even bypass a $W \oplus X$ memory policy, meaning ISR may offer better security guarantees than non-executable memory alone.

ISR implementations are generally software based, either using dynamic binary instrumentation or CPU emulation [13, 17]; we do not attempt to break that norm here. Hardware-based ISR schemes have also been proposed [15,20], although none have yet been applied to an x86 family of commodity CPUs. The randomization itself comes from code encryption, often with XOR schemes [7,8,13,17] or AES [12,15,20], and is done either at compile time or by modifying a precompiled binary. Because XOR encryption does not affect binary size, ISR schemes using this mechanism can randomize in-place at any time — even image load time, although this has not been widely explored. The decryption key is stored either in the binary itself [13] or in a database [17] and loaded into the process's address space along with the image. At process run time, instrumentation intercepts instruction fetches and decrypts them before sending them to the CPU's execution unit. Because unencrypted instructions should never exist in a process's address space, ISR can prevent both code injection and JIT-ROP attacks.

2.2 Protection against Code Injection

ISR protects against code injection attacks by invalidating shellcode during the derandomization process. In principle, to mount a code injection attack, the attacker must construct specific byte sequences corresponding to valid machine instructions and insert those bytes into the address space of the vulnerable process. An instrumented system, however, has changed the byte-to-instruction mapping, so what an attacker believes to be a valid sequence will in actuality be "derandomized" at runtime, resulting in an invalid shellcode and a likely segfault.

Consider the following byte sequence in vanilla x86-64, injected by an attacker as part of a shellcode, and its corresponding instructions:

```
55 48 89 e5  ||  push %rbp; mov %rsp,%rbp
```

Now, assume a binary instrumented with ISR, using an XOR encryption scheme with key `0x1757`. When the CPU fetches instructions from this shellcode, it will first decrypt them, then attempt to execute the following decrypted bytes:

```
42 1f 9e b2  ||  rex.X (bad); sahf; mov...
```

These garbage instructions will certainly not behave as the attacker intended, and will indeed likely crash the program when the first bad opcode is encountered.

2.3 Protection against ROP

ISR offers protection against ROP attacks by preventing attempts to read code and parse out gadget sequences. The first step of a ROP attack is always to identify what functionality is available in the target binary, either by manual pre-inspection or by disclosing code on-the-fly via a memory disclosure vulnerability [18,21]. However, in a randomized binary, attempts to read code from memory before it is decrypted will return garbled instruction sequences that cannot be interpreted as useful gadgets.[1] Therefore, memory disclosures in instrumented binaries cannot be used to construct a JIT-ROP exploit, since the contents of the leaked gadgets cannot be known without the decryption key.

Sinha et al. [20] rightly point out that an attacker can use a local copy of the targeted binary to identify gadgets ahead of time, and then simply direct the targeted program to the precomputed (encrypted) addresses. Under this model, an attack is possible; however, in keeping with ISR's heritage [13], we are primarily concerned with attacks against remote services and thus do not consider this vector within our threat model. Furthermore, variations in binary versions, compilation optimizations, and target system environments will further permute code locations on the target system and minimize the ability of an attacker to know exactly which (encrypted) gadgets are at which locations.

2.4 Threat Model

We assume the following threat model:

- The target system is protected with commodity defenses; namely, ASLR and a $W \oplus X$ memory policy.
- The adversary does not have access to the *exact* copy of any binary to be protected. This mitigates the attack vector described in § 2.3 where an attacker with no knowledge of specific code nevertheless creates a ROP payload from addresses alone.
- The target program may contain at least one memory disclosure vulnerability, and the adversary may use this

[1] Of course, the randomized bytes themselves could certainly be interpreted as gadgets — bytes are bytes — and an attacker can trivially construct a ROP sequence with them. Naturally, when executing this ROP sequence, the CPU will then decrypt the instructions and completely change the functionality of the exploit, if it functions at all.

vulnerability to read arbitrary memory and write to arbitrary memory.

- The adversary is remote and does not have physical access to the target system.

3 Implementation

In this section we describe our implementation of ISR using image encryption and dynamic binary instrumentation. Our toolchain is two steps: an encryption step using GNU binutils' `objcopy`, and a just-in-time decryption using Intel's Pin framework [14].

3.1 System Requirements

Our tool supports ELF binaries. We chose ELF because it is the most common executable format for Unix-like systems, it is targeted most by existing ISR implementations, and it completely separates code and data. Code and data separation is vital to correct encryption and decryption: mistakenly-encrypted data will never be decrypted (since it is not accessed as part of a CPU instruction fetch) and thus will corrupt the program. ISR implementations for Windows PE-format executables, which do not separate code and data, require an additional disassembly step between encryption and runtime to identify and extract data wrongly obfuscated as code [5].

Unlike Portokalidis et al.'s ISR implementation [17], we support 64-bit binaries and binaries with unaligned ELF sections. However, we too use a 16-bit XOR encryption scheme. Currently, our tool only supports statically linked binaries. This limitation is not present in other implementations and we aim to support shared libraries in the future. It is simply a matter of engineering effort to do so.

3.2 Encryption with `objcopy`

To encrypt target binaries, we modified the `objcopy` utility, v. 2.34. `objcopy`, part of the GNU binutils suite, transforms and rewrites object files for purposes of adding or removing headers or sections, augmenting precompiled code, or mere duplication. `objcopy` can parse and transform individual ELF sections, making it an ideal candidate for our purposes.

We modified the portion of `objcopy` responsible for duplicating ELF sections between files. As each section is about to be duplicated, we check if the section contains code and if it is to be loaded; if both conditions are true, we consider it a section for obfuscation. All sections to be obfuscated are then XOR'd with a random key before being copied to the new binary. Any section not to be obfuscated is copied verbatim, and the key is placed in a separate text file. Previous work has stored the key in an ELF header [13], but we break with this approach in order to enforce a clearer separation between obfuscated binary and decryption mechanism.

Although access to the exact (unencrypted) binary run by the target system is precluded our threat model, a pre-randomization step using `objcopy` provides an added benefit. Even if an adversary *did* have physical access to the target machine and could obtain the ELF file of the target binary, the file would be obfuscated and unreadable as long as the key was not known. Therefore, the attacker would still be unable to string together a complete ROP chain.

3.3 Decryption with Pin

We make use of Intel's Pin software [14] for our decryption environment. Pin is a powerful dynamic binary instrumentation (DBI) framework supporting all common architectures and operating systems. It provides numerous APIs for examining and instrumenting binaries at multiple granularities (image, section, routine, trace, and instruction), and "dynamic compilation" that combines instrumentation code and original binary instructions in a code cache. Program execution is then done from this cache.

Our so-called ISR decryption "pintool" begins by reading the decryption key from a given file. We then register a callback to occur on each instruction fetch. When the CPU requests an instruction, we load it from our encrypted binary, XOR it with the supplied key, and provide the result back to the CPU.

Our tool must contend with Linux's Virtual Dynamic Shared Object (VDSO), a read-only library injected into every process to efficiently handle certain low-impact syscalls. Although our implementation currently only supports statically linked binaries, this shared library is present in the address space of, and used in, our instrumented programs. Because the VDSO is unencrypted, however, attempts to fetch and decrypt instructions from it inadvertently garble these instructions — an example of ISR's security guarantees working against it. To avoid this, we instruct Pin to search `/proc/self/maps` for the VDSO's address range before running the instrumented binary. Any instruction fetches falling within this range are copied and returned without decryption, allowing the VDSO's code to be executed normally. Because the VDSO is read- and execute-only, it is safe to execute uninstrumented code from it since there is no way an adversary can modify it without first mounting some other type of attack.[2]

4 Evaluation

In this section we evaluate our ISR implementation for three metrics: correctness, performance, and security. We tested our implementation against GNU coreutils v. 8.32, using a heavily-modified version of coreutils' built-in test suite. Coreutils were statically compiled and linked using musl libc v.

[2] Granted, this unencrypted library could be used to create ROP chains. A simple extension to our ISR implementation could encrypt the VDSO's code at load time before the target program begins.

Binary Type	Expire	Pass	Fail	Skip	Test Err
uninstrumented	0	383	22	109	5
blank pin	2	267	122	113	14
encryption	2	268	121	113	14

Table 1: Results of correctness tests. Tests were killed and marked expired after 5 minutes. "Test Err" refers to a problem with the test framework, not the test itself.

1.2.0 [3], an alternative libc implementation, because glibc does not support static linkage. Binaries were encrypted using our modified `objcopy` and wrapped with a shell script that invoked our pintool. Our test machine was equipped with a 6th generation Intel Xeon E3-1240 CPU at 3.5 GHz and 16 GB of RAM, running Debian 10 based on Linux kernel 4.19.

4.1 Modification of coreutils' test suite

The test suite provided with coreutils is designed to verify compilation and installation [9], not to benchmark the system, which resulted in many necessary modifications to work with our ISR implementation. Specifically, we modified the suite to remove a code generation step that took place before tests were run and which overwrote our encrypted binaries. We also modified the tests themselves to be run directly as shell scripts, rather than as inputs to a test runner. Additionally, coreutils tests modify the system's PATH, rather than use absolute program addresses. Therefore, any test that required an additional coreutil program (for example, an `ls` test that requires `basename`) invoked our instrumented version of the other program. We modified these tests such that when external programs were required, wherever possible, pre-installed versions on the system were used. Finally, we wrote a custom test harness in Python to more accurately test, benchmark, and record the performance of the coreutils tests. The Python harness supports both Perl and Bash scripts. However, Perl-based tests were excluded from our analysis because they were incompatible with our wrapper scripts and reported a disproportionate number of false negatives.

4.2 Correctness Tests

To verify correctness, each coreutils test was run three times. First, an uninstrumented version of the binary was tested to establish a baseline for comparison. Then, we tested a version which was unencrypted but instrumented with an empty pintool. The tool, which just runs the underlying program, was used to identify any failures caused by Pin itself and distinguish those from any errors introduced by the ISR implementation, which we tested third.

Table 1 gives results of the correctness tests, which demonstrate that our ISR implementation is sound. In all categories,

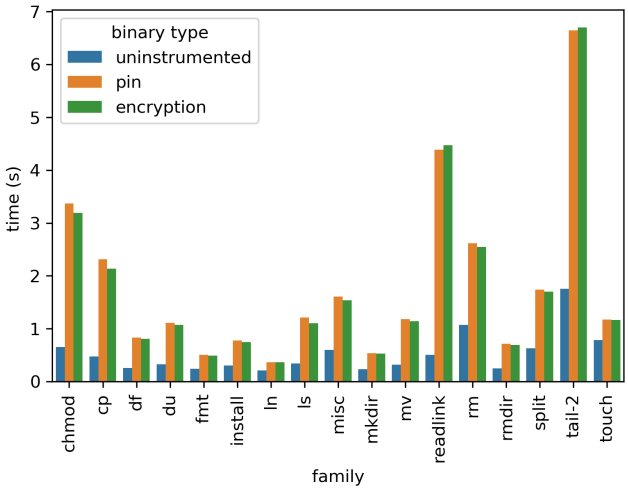

Figure 1: Performance results. Figure reports average execution time of all tests within a family, each run 10 times. Tests that did not pass under all conditions were excluded as outliers.

our ISR implementation matches or improves upon the correctness of an empty pintool. These results clearly indicate that, for a wide range of programs affecting different system functionality, no errors are introduced by our ISR implementation.

Many of the tests that failed under both the blank pintool and ISR did so because of our binary wrapper script. The tests invoked our wrapper, which in turn started the Pin instrumentation in a new process with a different ID and name. Therefore, tests which expected certain process information were given data that appeared to be incorrect. Correcting these errors is simply a matter of engineering effort to rewrite the tests themselves, which we did not undertake because it did not affect the correctness of our ISR implementation.

Tests were skipped for a number of reasons. Many were incompatible with a network filesystem, with which our test machine was equipped, or required root access, which we did not have. Additionally, while we successfully removed the coreutils' tester's code generation step, some tests themselves generated code that expected a program loader and thus was incompatible with statically linked binaries. These issues were detected ahead of time, and the affected tests were not run.

4.3 Performance Tests

To benchmark performance, each test was run ten times, again under three different conditions: uninstrumented, blank pintool, and ISR pintool. Tests were bucketed into "families" designated by the coreutils program they test, with one miscellaneous bucket. Execution time was grouped by family and

averaged. Importantly, we only considered performance of tests which passed in all three conditions. We observed many tests which passed when uninstrumented but failed under a blank pintool, after taking significantly longer to complete. To prevent these failures from increasing the mean test completion time, we removed them as outliers from our analysis.

Figure 1 reports benchmarking results and demonstrates that while the use of a pintool incurs significant overhead, our ISR implementation is highly efficient. Specifically, we report performance overheads ranging between 48.73% and 790.65% when compared against an uninstrumented baseline, with an average of 228.61%. However, when we compare our ISR implementation against the empty pintool, we observe an average overhead of just 3.42%.

The performance decrease associated with the use of a pintool results from the startup time of the tool. Short-lived utilities such as coreutils are disproportionately affected by this startup cost, since it represents a much larger percentage of their lifetime than it would for a long-lived web server. We have explored additional optimizations to further improve the performance of our pintool. Specifically, we look to the use of compiler prefetching suggestions to aid the CPU's branch predictor. In later work, we hope to implement these and other optimizations, as well as benchmark performance against a longer-lived program.

4.4 Security Tests

To test the security guarantees of our ISR implementation, we created a toy program to read arbitrary memory of an arbitrary size from an arbitrary location — a hacker's dream. We also equipped the program with an algorithm to detect the address of the `mprotect` system call in its address space. We chose `mprotect` because it is a function commonly targeted by adversaries seeking to loosen restrictions on blocks of the address space in order to write or execute malicious code, but our algorithm would have behaved similarly for any function. In line with our adversary model, the program was compiled position-independent to enable ASLR and obeyed a $W \oplus X$ policy. While we did perform our security analysis locally, the program would have functioned identically on a remote system. We did not need to examine the binary to perform our analysis.

We ran our test program twice, once without instrumentation and once encrypted with ISR. When the uninstrumented program was run, it immediately and accurately found the address of `mprotect`. This was repeated several times, with the image's code location permuted each time. However, when the encrypted binary was run under our pintool, the program terminated correctly but without finding the address of `mprotect` anywhere in the code section. This was expected, since the function's bytes were randomized and thus could not be positively identified. The result of this test demonstrates that our ISR implementation can successfully prevent JIT-ROP style attacks where the adversary has no advance knowledge of the binary itself, despite the presence of an extremely liberal memory disclosure vulnerability.

5 Previous and Related Work

A large body of research is dedicated to ISR. As part of their original proposal, Kc et al. [13] modify the bochs x86 emulator to simulate ISR on a single-system image with just a few instrumented programs. Boyd et al. [8] repeat this study, but improve instrumentation performance with three heuristics: probabilistic identification of functions susceptible to code-injection attacks based on their call-graph distance to input functions, identification of program vulnerabilities using a separately-instrumented honeypot machine, and static analysis of program source code. Barrantes et al. [7] present the only ISR implementation to scramble code at load time, using the Valgrind emulator, and Hu et al. [12] use more secure AES encryption with the Strata DBI framework. Portokalidis et al. [17] offer an XOR-encrypted, Pin-based approach, the first to support shared libraries. ASIST [15] is the first ISR solution to run natively in hardware with negligible overhead, using a SPARC processor on an FPGA board, and the first to instrument the Linux kernel. Most recently, Sinha et al. [20] have combined the ASIST approach with fine-grained code randomization and an AES encryption key specifically to thwart JIT-ROP attacks, and support encrypting an entire system from bootloader to user applications.

Another program hijacking defense is control-flow integrity (CFI) [4]. CFI inspects the target of any control-flow transfer within a program to ensure that it occurs along the edge of a predefined *control-flow graph*. If the transfer does not follow such an edge, it is not part of normal program execution and likely erroneous or malicious. However, CFI is difficult to implement if source code is not available, and Carlini et al. [10] have shown that even under the most restrictive CFI policy, hijacking is still possible. Nevertheless, the Chromium browser and parts of the Android OS have introduced forward-edge CFI in production [1, 2].

A third area of research in preventing code reuse attacks is execute-only memory. Under such a scheme, the system forbids read accesses from any memory page containing code. This can be done by marking code pages as not present unless accessed as part of an instruction fetch [6] or by using a thin hypervisor and related hardware features to prevent such reads [11]. Execute-only memory relies on fine-grained ASLR to permute code layout and prevent precomputation of ROP chains but aims to fix the vulerability described in § 1 by removing the impact of a memory disclosure bug. A related technique, destructive code reads [22], allows reading executable memory, but immediately mangles the memory after it is read and renders it subsequently useless as code.

These defenses, and many others, aim to solve the same software security problems as ISR with different approaches.

6 Conclusion

We present an updated implementation of ISR based on Intel's Pin DBI framework, which seeks to protect binaries against JIT-ROP attacks by encrypting code with a 16-bit XOR key to neutralize memory disclosure vulnerabilities that could otherwise allow for arbitrary code reads. Our tool does not require hardware or compiler modifications, and can be done in-place with a precompiled binary using existing open-source tools. We test and benchmark our tool against the GNU coreutils suite of programs. While the total overhead of our tool is significant, we report an average 3.42% penalty over the use of a pintool alone.

Despite the performance penalties incurred by Pin instrumentation, we consider our tool a successful implementation. In future work, we hope to support shared libraries, larger keys, memory protection of the instrumentation itself, and further performance optimizations.

References

[1] Control Flow Integrity. https://source.android.com/devices/tech/debug/cfi.

[2] Control Flow Integrity - The Chromium Projects. https://www.chromium.org/developers/testing/control-flow-integrity.

[3] musl libc. https://musl.libc.org/.

[4] Martin Abadi, Mihai Budiu, and Úlfar Erlingsson. Control-Flow Integrity. November 2005.

[5] Muhammad Akbar. ISRuPIN-Win, 2020. https://github.com/makbar/ISRuPIN-Win.

[6] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1342–1353, Scottsdale, Arizona, USA, November 2014. Association for Computing Machinery.

[7] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 281–289, Washington D.C., USA, October 2003. Association for Computing Machinery.

[8] Stephen W. Boyd, Gaurav S. Kc, Michael E. Locasto, Angelos D. Keromytis, and Vassilis Prevelakis. On the General Applicability of Instruction-Set Randomization. *IEEE Transactions on Dependable and Secure Computing*, 7(3):255–270, July 2010. Conference Name: IEEE Transactions on Dependable and Secure Computing.

[9] Pádraig Brady. How the GNU coreutils are tested, January 2017.

[10] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. pages 161–176, 2015.

[11] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780, May 2015. ISSN: 2375-1207.

[12] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 2–12, Ottawa, Ontario, Canada, June 2006. Association for Computing Machinery.

[13] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 272–280, Washington D.C., USA, October 2003. Association for Computing Machinery.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*, 40(6):190–200, June 2005.

[15] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. ASIST: architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 981–992, Berlin, Germany, November 2013. Association for Computing Machinery.

[16] PaX Team. PaX address space layout randomization. Technical report, March 2003.

[17] Georgios Portokalidis and Angelos D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10,

pages 41–48, Austin, Texas, USA, December 2010. Association for Computing Machinery.

[18] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, Alexandria, Virginia, USA, October 2007. Association for Computing Machinery.

[19] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, Washington DC, USA, October 2004. Association for Computing Machinery.

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

[21] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588, May 2013. ISSN: 1081-6011.

[22] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 256–267, Denver, Colorado, USA, October 2015. Association for Computing Machinery.