Securing Legacy Software against Real-World Code-Reuse Exploits: Utopia, Alchemy, or Possible Future?

Ahmad-Reza Sadeghi, Lucas Davi
Technische Universität Darmstadt, Germany and
Intel Collaborative Research Institute
for Secure Computing

Per Larsen
University of California, Irvine

ABSTRACT

Exploitation of memory-corruption vulnerabilities in widely-used software has been a threat for over two decades and no end seems to be in sight. Since performance and backwards compatibility trump security concerns, popular programs such as web browsers, servers, and office suites still contain large amounts of untrusted legacy code written in error-prone languages such as C and C++. At the same time, modern exploits are evolving quickly and routinely incorporate sophisticated techniques such as code reuse and memory disclosure. As a result, they bypass all widely deployed countermeasures including data execution prevention (DEP) and code randomization such as address space layout randomization (ASLR).

The good news is that the security community has recently introduced several promising prototype defenses that offer a more principled response to modern exploits. Even though these solutions have improved substantially over time, they are not perfect and weaknesses that allow bypasses are continually being discovered. Moreover, it remains to be seen whether these prototype defenses can be matured and integrated into operating systems, compilers, and other systems software.

This paper provides a brief overview of current state-of-the-art exploitation and defense techniques against run-time exploits and elaborates on innovative research prototypes that may one day stem the tide of sophisticated exploits. We also provide a brief analysis and categorization of existing defensive techniques and ongoing work in the areas of code randomization and control-flow integrity, and cover both hardware and software-based solutions.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*

Keywords

software exploitation; fine-grained randomization; control-flow integrity

1. INTRODUCTION

When the C language was designed four decades ago, computers were much less powerful, less networked, and thus much less exposed to malicious activities. In the pursuit of efficiency and flexibility, security features such as automatic memory management, strong typing, and overflow checks were omitted from C. As a result, programming errors can lead to memory corruption that causes unexpected program behavior and can be exploited for malicious purposes.

Modern systems incorporate a large amount of C/C++/Objective-C code and hand-written assembly code for performance and legacy reasons. This includes operating system kernels, web browsers, document viewers and language implementations for JavaScript, Flash, Java, PHP, etc. Despite the existence of many type-safe languages, members of the C family consistently inhabit the top ten in rankings of programming language popularity [54, 44]. This leaves the security research community with the challenge of mitigating exploits without abandoning the unsafe language features or introducing high overheads while at the same time remaining fully compatible with all existing code.

The goal of this paper is to give an overview of recent offensive and defensive systems security research in the context of runtime exploits. We start by giving a general overview of code reuse attacks (Section 1.1) and defenses (Section 1.2) and then describe offensive techniques that threaten to undermine the security of several improved defenses (Section 2). Finally, Section 3 highlights advanced defenses that aim to resist offensive techniques while at the same time offering practicality and efficiency.

1.1 From Code Injection to Code Reuse

The classic buffer overflow vulnerability allows an adversary to inject malicious code on the stack and overwrite a function's return address in order to direct execution to the exploit payload [3]. Stack smashing attacks became rare after deployment of techniques such as stack canaries [14]. However, attackers quickly adapted and moved to exploitation on the heap [42]. Modern operating systems defend against code injection attacks through Data Execution Prevention (DEP) [41]. DEP prohibits memory pages from being both writable and executable thereby preventing an adversary from injecting and directly executing code.

While standard defenses have all but obsoleted some types of attacks, new attacks have appeared in their place. In particular, code reuse emerged in response to the widespread deployment of DEP as it turns out that injecting malicious data is just as effective as malicious code injection [38]. The most popular variant of code reuse, return-oriented programming (ROP), reuses short code sequences, called gadgets, terminated by a return instruction (or another indirect branch) [48, 12]. ROP has been applied to many processor architectures: SPARC [9], Atmel AVR [22], PowerPC [35], and ARM [33]. In addition, ROP has been shown to be Turing-complete meaning that an adversary can generate arbitrary malicious execution.

1.2 Mitigating Code Reuse

We briefly introduce the most prominent techniques to prevent code reuse: code randomization and control-flow integrity.

Code randomization such as Address Space Layout Randomization (ASLR) can in principle prevent ROP attacks by making the location of gadgets unpredictable. Today, ASLR is enabled on nearly all modern operating systems including Windows, Linux, iOS, or Android. For the most part, current ASLR implementations randomize the base (start) address of segments such as the stack, heap, libraries, and the executable itself between consecutive runs of the application. The goal is to force adversaries to guess the location of the functions and instruction sequences needed to successfully launch a code reuse attack.

Unfortunately, ASLR suffers from two main problems: first, the entropy on 32-bit systems is too low, and thus ASLR is vulnerable to brute-force attacks [49]. Second, ASLR is highly vulnerable to *memory disclosure* attacks [47] where the adversary gains knowledge of a single runtime address and uses that information to infer the memory layout of the application.

To thwart these attacks, a number of fine-grained ASLR and code randomization schemes have been recently proposed [7, 32, 39, 28, 55, 30, 29]. The underlying idea in these works is to randomize the data and code structure, for instance, by shuffling functions or basic blocks (ideally for each program run [55]). The assumption underlying all these works is that the disclosure of a single address no longer allows an adversary to launch a code reuse attack. However, as will be explained in Section 2.1, more involved types of memory disclosure vulnerabilities can be exploited to bypass fine-grained code randomization.

Control-flow integrity (CFI) is another promising exploit mitigation mechanism [1, 2]. The main idea of CFI is to compute an application's control-flow graph (CFG) prior to execution, and then monitor its runtime behavior to ensure that the control-flow follows a legitimate path in the CFG. Any deviation from the CFG leads to a CFI exception and subsequent termination of the application.

Validating all indirect control-flow transfers can have a substantial performance impact that prevents widespread deployment. For instance, when validating function returns using a shadow stack, the average overhead of CFI can be as high as 21% [1] on average. Consequently, several CFI frameworks have been proposed that tackle the practical shortcomings of the original CFI approach. ROPecker [13] and kBouncer [40], for example, leverage the branch history table of modern x86 processors to perform a CFI check on a short history of executed branches. Zhang and Sekar [58] and Zhang et al. [57] applied coarse-grained CFI policies using binary rewriting to protect COTS binaries. In section 2.2, we will discuss the security implications of CFI policies that trade off security for efficiency.

2. ADVANCED ATTACKS

In this section we present two recent attack techniques that undermine modern exploit mitigation techniques. First, we present JIT-ROP, a just-in-time code-reuse attack that bypasses defenses that are based on randomizing the code layout of an application (Section 2.1). Second, we present an attack technique that bypasses several control-flow integrity approaches (Section 2.2). These attacks demonstrate weaknesses of two prominent exploit mitigation techniques, namely code randomization and control-flow integrity. They also motivate several improved mitigation techniques that we discuss in Section 3.

2.1 Just-In-Time Code Reuse

Just-in-time return-oriented programming (JIT-ROP) circumvents fine-grained ASLR by finding gadgets and generating the ROP payload at runtime using the scripting environment of the target application (e.g., a browser or document viewer). As with many real-world ROP attacks, the disclosure of a *single* runtime memory address is sufficient. However, in contrast to standard ROP attacks, JIT-ROP does not require the precise knowledge of the code part or function the memory address points to. It can use any code pointer such as a return address on the stack to instantiate the attack. Based on that leaked address, JIT-ROP discloses the contents of other memory pages by recursively searching for pointers to other code pages and generates the ROP payload at runtime.

The workflow of a JIT-ROP attack is shown in Figure 1. Here, we assume that fine-grained ASLR has been applied to each executable module in the address space of the (vulnerable) application. First, the adversary exploits a memory disclosure vulnerability to retrieve the runtime address of a code pointer ❶. One of the main observations of Snow et al. [51] is that the disclosed address will reside on a 4KB-aligned memory page (Page_0 in Figure 1). Hence, at runtime, one can identify the start and end of Page_0 ❷. Using a disassembler at runtime, Page_0 is then disassembled on-the-fly ❸. The disassembled page provides 4KB of gadget space ❹, and more importantly, it is likely that it contains direct branch instructions to other pages, e.g., a call to Func_B ❺. Since Func_B resides on another memory page (namely Page_1), JIT-ROP can again determine the page start and end, and disassemble Page_1 ❻. This procedure is repeated as long as new direct branches pointing to yet undiscovered memory pages can be identified ❼. Using the disassembled pages, a runtime gadget finder is then used to identify useful ROP gadgets (e.g., LOAD, STORE, or ADD ❽). Finally, the ROP payload is composed based on the discovered ROP gadgets and a high-level description of the desired functionality provided by the adversary ❾.

The threat of memory disclosure is not limited to JIT-ROP. For example, Shacham et al. [49], Bittau et al. [8] and Siebert et al. [50] all demonstrated how attackers can re-

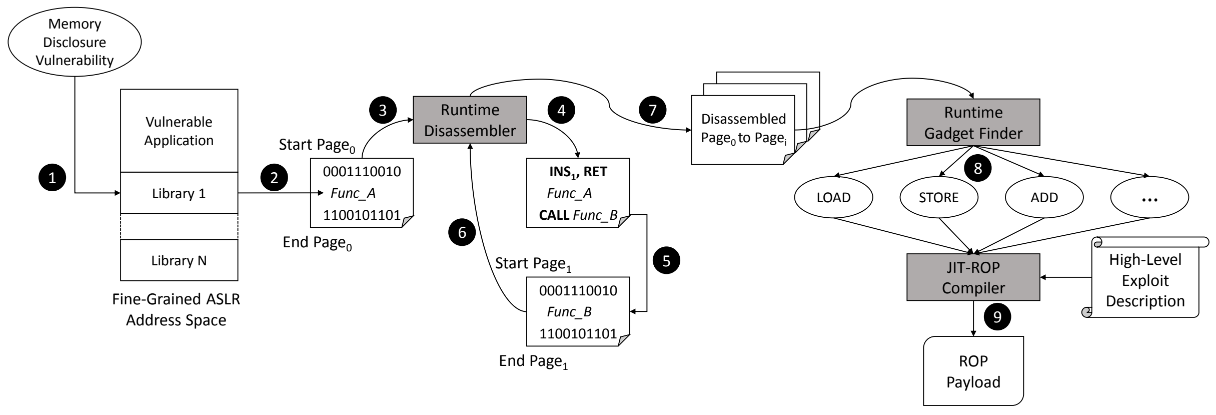

Figure 1: High-level overview of a JIT-ROP attack [51].

peatedly probe servers and analyze response characteristics to remotely bypass both coarse and fine-grained ASLR.

2.2 Bypassing Coarse-Grained CFI

The benefits of the so-called coarse-grained solutions come at the price of relaxing the original CFI policy. Abstractly speaking, coarse-grained CFI allows for CFG relaxations that allow many additional execution paths beyond those intended by the programmer. The most notable difference is that the coarse-grained CFI policy for return instructions only validates if the return address points to an instruction that follows directly after a call instruction [40, 58]. In contrast, the original policy for fine-grained CFI ensures that the return address points to the original caller of a function (based on a shadow stack). That is, a function return is only allowed to return to its original caller. On the other hand, several solutions use heuristics to compensate for the coarse-grained protection of returns, e.g., by monitoring the number of instructions executed between a pre-defined number of consecutive indirect branches [40, 13]. The goal is to detect the execution of a number of short instruction sequences which is a typical pattern of return-oriented programming attacks.

Davi et al. [18] conduct a systematic security analysis of the recently proposed CFI solutions including kBouncer [40], ROPecker [13], CFI for COTS binaries [58], ROPGuard [23], and Microsofts' EMET tool [36]. In particular, they derive a combined CFI policy that, for each type of indirect indirect branch and behavioral heuristic (e.g., the number of instruction executed between two indirect branches), uses the most restrictive setting among the aforementioned policies. Their security analysis demonstrates that based on the access to only a *single* — and commonly used system library in Windows — an adversary can still construct a Turing-complete gadget set. In particular, a new gadget type has been developed called *long-NOP*. This gadget bypasses heuristics that check for chains of short instruction sequences by invoking a long sequence that performs well-controlled memory writes without disrupting the actual payload, i.e., it also compensates for side-effects by saving registers before it is executed and loading them after the memory writes have been issued.

Other researchers have demonstrated weaknesses of coarse-grained CFI by constructing attacks that target specific CFI implementations. Göktas et al. [26] demonstrate attacks against the Compact Control-Flow Integrity and Randomization (CCFIR) [57] approach using call-preceded gadgets to invoke sensitive functions via direct calls. Carlini and Wagner [11] as well as Schuster et al. [46] demonstrate flushing attacks that eliminate return-oriented programming traces before a critical function is invoked.

3. ADVANCED DEFENSES

We now turn our attention to approaches that aim to withstand even the advanced exploitation techniques outlined in the previous section.

3.1 Fine-Grained Code Randomization

The introduction of JIT-ROP and other offensive techniques relying on memory disclosure, made it apparent that increasing the granularity of code randomization is not enough to thwart runtime exploits. There are two ways that defenders can address the threat of memory disclosure: 1) taking steps to prevent memory disclosure or 2) tolerating memory disclosure. We discuss each strategy in turn.

Hiding the code layout is one way to prevent adversaries from launching JIT-ROP attacks against diversified code. Backes and Nürnberger [6] proposed to prevent code discovery (Step ❼) in the original JIT-ROP approach by hiding pointers between code pages. Davi et al. [19] show that this approach is insecure because the adversary can harvest code pointers from C++ virtual tables instead of following pointers between code pages. Backes et al. [5] and Gionta et al. [25] later proposed two execute-only memory solutions: XnR and HideM respectively. However, these two approaches do not consider the threat of pointer harvesting.

Readactor by Crane et al. [16] aims at addressing some of the shortcomings of previous approaches. Specifically, it combines hardware-enforced execute-only memory and code pointer hiding to prevent both *direct* memory disclosure (code reads) and *indirect* disclosure through code pointer harvesting. In addition, Readactor scales beyond benchmarks to complex, real-world software such as browsers and JIT engines. One remaining challenge is that Readactor only runs on processors that support hardware accelerated paging[1] (HAP) and requires an operating system or kernel extension to exposes these features to application programs.

[1]Intel calls this feature Extended Page Tables while AMD markets this features as Nested Page Tables.

Control-flow randomization is an alternative way to tolerate code layout disclosure. Whereas Readactor, XnR, and HideM aim to *prevent* memory disclosure the Isomeron approach by Davi et al. [19] instead *tolerates* memory disclosure. A successful ROP attack requires an attacker to, among other things, 1) discover gadget addresses and 2) reliably transfer the control flow from one gadget to another. Most probabilistic defenses target the first requirement. Isomeron targets the second and therefore operates on the conservative assumption that adversaries know the code layout and instead randomizes control-flow transfers. In particular, Isomeron keeps two isomers (clones) of all functions in memory; one isomer retains the original program layout while the other is diversified. On each function call, Isomeron randomly determines whether the return instruction should switch execution to the other isomer or keep executing functions in the current isomer. Upon each function return, the result of the random trial is retrieved, and if a decision to switch was made, an offset (the distance between the calling function f and its isomer f') is added to the return address. Since the attacker does not know which return addresses will have an offset added and which will not, return addresses injected during a ROP attack will no longer be used "as is" and instead, the ROP attack becomes unreliable due to the possible addition of offsets to the injected gadget addresses. Since Isomeron is implemented as dynamic binary instrumentation framework it suffers from performance penalties and requires extra memory to hold the two program instances. In order to bring the performance overhead in line with efficient code layout randomization techniques [29, 20, 55], Isomeron could be integrated into a compiler.

Isomeron may be able to simultaneously protect against side-channel attacks in addition to code-reuse attacks. We observe that Crane et al. uses a similar control-flow randomization mechanism to rapidly alternate between two diversified program clones to thwart cache-based side-channel attacks [15]. We believe the two approaches can be readily combined.

3.2 Control-Flow Integrity

The literature on control-flow integrity is substantial. Consequently, we limit our discussion to recent and unconventional approaches.

CFI policy randomization prevents attacks against coarse-grained CFI. Opaque Control-Flow Integrity (O-CFI) is a hybrid approach that recasts CFI as a bounds checking problem [37]. Instead of checking the target address of each indirect branch against a list of valid targets, O-CFI checks that the target address is within the bounds defined by the maximal and minimal addresses in the list of valid targets. To avoid attacks wherein attackers construct a ROP chain that adheres to all bounds checks, O-CFI randomizes the code layout at load-time. Code layout randomization has the desirable side-effect of also randomizing the bounds that each ROP chain must adhere to. By protecting the bounds table against memory leakage vulnerabilities, O-CFI aims to resist JIT-ROP [51], Blind ROP [8], and gadget-stitching attacks [18, 26, 11, 27]. With an average overhead of 4.7%, O-CFI offers efficient protection on current processors. However, forthcoming Intel processors will include Memory Protection eXtensions (MPX) with support for hardware-accelerated bounds checking. Note that like Isomeron, O-CFI randomizes the code layout but *tolerates* code layout disclosure. Where Isomeron randomizes the control-flow, O-CFI constrains the control-flow using a random and thus unknown control-flow policy—in both cases, the adversary is unable to reliably chain ROP gadgets together. The current implementation of O-CFI is based on binary rewriting which makes it hard to recover a precise control-flow graph. A source-based implementation of O-CFI would be able to impose tighter bounds on indirect control-flow transfers (particularly for programs written in C++) and thus increase resilience to attacks.

Forward-edge CFI: In general, one can distinguish between backward and forward edges in the control-flow graph (CFG) of an application. The former are representative for function return instructions. In contrast, the latter edges are due to function call and jump instructions. Recently, a number of approaches have focused on applying CFI to forward-edges. In particular, compilers for C++ applications emit indirect call instructions to support virtual method calling via virtual tables. SAFEDISPATCH [31] and Google's CFI compiler [53] both ensure that an adversary cannot manipulate a virtual table (vtable) pointer so that it points to an adversary-controlled (malicious) vtable. However, these approaches require the source code of the application which might not be always readily available. In order to protect binary code, a number of CFI approaches have been presented recently [24, 56, 43]. Although these approaches require no access to source code, they are not as fine-grained as their compiler-based counterparts. A novel attack technique denoted as COOP (counterfeit object-oriented programming) undermines the CFI protection of these binary instrumentation-based defenses by invoking a chain of virtual methods through legitimate call sites to induce malicious program behavior [45].

Hardware-assisted CFI: The majority of research on CFI has focused on software-based solutions. However, hardware-based CFI approaches have several advantages over software-based counterparts: first, it is significantly more efficient. Second, CFI support in hardware easily allows compilers to harden software programs as only single CFI instructions need to be emitted rather than complex and large CFI checking code. Third, dedicated CFI hardware instructions and separate CFI memory provide an efficient and strong protection of critical CFI control-flow data.

The first hardware-based CFI approach has been presented by Budiu et al. [10]. They realized the original CFI proposal [1, 2] as a CFI state machine in a simulation environment of the Alpha processor. HAFIX is a recent hardware-based control-flow integrity protection that has been implemented on real hardware targeting Intel Siskiyou Peak and SPARC [17, 4]. It generates 2% performance overhead across different embedded benchmarks and focuses on thwart return-oriented programming attacks exploiting function returns.

As previously mentioned, the original software-based CFI implementation validates function returns using a shadow stack [2]: all return addresses that are pushed on the program's stack through call instructions are copied to a protected shadow stack. However shadow stack significantly decrease performance and lead to false positives for certain programming constructs (e.g., C++ exceptions with stack unwinding, setjmp/longjmp).

To reduce the cost of maintaining a shadow stack, HAFIX simply confines function returns to active call sites [17]. In other words, it forces a return to target a call-preceded instruction inside a function that is currently executing. This CFI policy can be efficiently implemented in hardware and requires only minimal changes to the compiler.

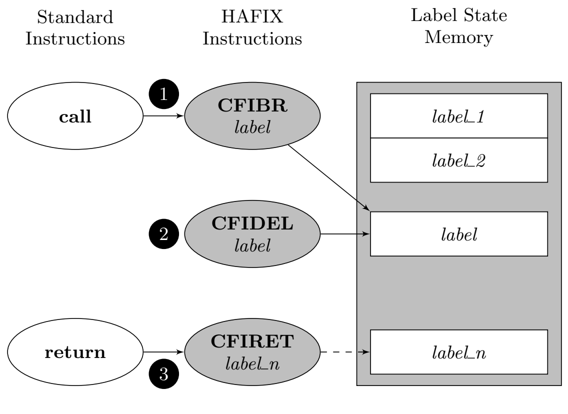

Figure 2: Abstract design of HAFIX

Figure 2 shows the underlying design to enforce the HAFIX CFI policy. To monitor functions that are currently executing, HAFIX requires the compiler to assign unique labels to each function. Further, it forces each subroutine upon call to issue a `CFIBR` instruction. This instruction loads the label of the function into a dedicated memory area, called the *label state memory*, to indicate that the function is active (Step ❶). Internally, direct and indirect call instructions lead to a processor state switch in which the processor only accepts `CFIBR`. To deactivate a function, HAFIX uses the `CFIDEL` instruction which effectively removes the label from the label state memory (Step ❷).

In HAFIX, return instructions need to target an active call site. This is enforced by switching the processor state where only the so-called `CFIRET` instruction is allowed. In this state, only a `CFIRET` that uses a currently active label in the label state memory is allowed (Step ❸). Otherwise, a CFI exception is raised.

On the compiler side, HAFIX only requires the compiler to emit the new CFI instructions at their corresponding places: `CFIBR` at function start, `CFIRET` at all call sites, and `CFIDEL` at function return.

HAFIX reduces the number of available call sites by 80% for static benchmark binaries. In other words, only every fifth call site is a valid target of a return. Obviously, the gadget space is further reduced for applications that link to shared libraries since only the instruction following the call of a library function is a valid target for the return.

Currently, HAFIX targets bare-metal code for static applications running on embedded processors. It remains to show how HAFIX can be applied to complex applications that also link to shared libraries and run on top of a operating system.

3.3 Code-Pointer Integrity

Based on a careful and detailed analysis of memory corruption [52], Szekeres et al. identified code-pointer integrity (CPI) as a defensive alternative to both code randomization and control-flow integrity. Whereas CFI seeks to detect invalid targets of control-flow transfers, CPI places sensitive control-flow data (e.g., function pointers and return addresses) in a "safe region" separated from all non-control-flow data. Kuznetsov et al. [34] presented the first implementation of code-pointer integrity. Although the worst-case overheads of CPI are high, the overall cost of protection is competitive with CFI and code randomization. CPI is no silver bullet, however, and bypasses have started to appear [21].

4. CONCLUSION

While we have barely skimmed the surface of modern offensive and defensive research, we hope to have brought some clarity to the exciting field of runtime exploits. We do not anticipate that any of the advanced defenses we have discussed will put an end to exploitation overnight. Instead, we expect slow but steady progress towards mature software-based defenses, slower progress towards support for hardware security mechanisms, and glacially slow progress towards safer languages. It is evident that building secure systems that contain large amounts of unsafe legacy code is a non-trivial if not impossible undertaking.

5. REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13:4:1–4:40, 2009.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), 1996. `http://www.phrack.org/issues.html?id=14&issue=49`.

[4] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. HAFIX: Hardware-assisted flow integrity extension. In *Design Automation Conference*, DAC '15, 2015.

[5] 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 *ACM Conference on Computer and Communications Security*, CCS '14, 2014.

[6] M. Backes and S. Nürnberger. Oxymoron - making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, 2014.

[7] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[8] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy*, S&P '14, 2014.

[9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *ACM Conference on Computer and Communications Security*, CCS '08, 2008.

[10] M. Budiu, U. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, 2006.

[11] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[12] S. Checkoway, L. V. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security*, CCS '10, 2010.

[13] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Network And Distributed System Security Symposium*, NDSS '14, 2014.

[14] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, D. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[15] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Network And Distributed System Security Symposium*, NDSS '15, 2015.

[16] 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 *IEEE Symposium on Security and Privacy*, S&P '15, 2015.

[17] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Design Automation Conference - Special Session: Trusted Mobile Embedded Computing*, DAC '14, 2014.

[18] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[19] 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 *Network And Distributed System Security Symposium*, NDSS '15, 2015.

[20] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *ACM Symposium on Information, Computer and Communications Security*, ASIACCS '13, 2013.

[21] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point: On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy*, S&P '15, 2015.

[22] A. Francillon and C. Castelluccia. Code injection attacks on Harvard-architecture devices. In *ACM Conference on Computer and Communications Security*, CCS '08, 2008.

[23] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. `http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf`, 2012.

[24] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference*, ACSAC '14, 2014.

[25] J. Gionta, W. Enck, and P. Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *ACM Conference on Data and Application Security and Privacy*, CODASPY '15, 2015.

[26] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, S&P '14, 2014.

[27] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium*, 2014.

[28] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *IEEE Symposium on Security and Privacy*, S&P '12, 2012.

[29] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automatic software diversity. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '13, 2013.

[30] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Diversifying the software stack using randomized NOP insertion. In S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense II*, volume 100 of *Advances in Information Security*. Springer New York, 2013.

[31] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Network And Distributed System Security Symposium*, NDSS '14, 2014.

[32] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference*, ACSAC '06, 2006.

[33] T. Kornau. Return-oriented programming for the ARM architecture. Master's thesis, Ruhr University Bochum, Germany, 2009.

[34] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, 2014.

[35] F. Lindner. Router exploitation. `http://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-SLIDES.pdf`, 2009.

[36] Microsoft. Enhanced Mitigation Experience Toolkit. `https://www.microsoft.com/emet`.

[37] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Network And Distributed System Security Symposium*, NDSS '15, 2015.

[38] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11(58), 2001. `http://www.phrack.org/issues.html?issue=58&id=4`.

[39] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In

IEEE Symposium on Security and Privacy, S&P '12, 2012.

[40] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.

[41] PaX. *Homepage of The PaX Team*, 2001. `http://pax.grsecurity.net`.

[42] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security & Privacy*, pages 20–27, 2004.

[43] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Network and Distributed System Security Symposium*, NDSS '15, 2015.

[44] The redmonk programming language rankings, 2015. `http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/`.

[45] 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 *IEEE Symposium on Security and Privacy*, S&P '15, 2015.

[46] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '14, 2014.

[47] F. J. Serna. The info leak era on software exploitation. In *Black Hat USA*, 2012.

[48] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*, CCS '07, 2007.

[49] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*, CCS '04, 2004.

[50] J. Siebert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM Conference on Computer and Communications Security*, CCS '14, 2014.

[51] K. Z. Snow, F. Monrose, L. V. 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 *IEEE Symposium on Security and Privacy*, S&P '13, 2013.

[52] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: eternal war in memory. In *IEEE Symposium on Security and Privacy*, S&P '13, 2013.

[53] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[54] Tiobe programming community index, 2015. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[55] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, 2012.

[56] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *Network and Distributed System Security Symposium*, NDSS '15, 2015.

[57] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *IEEE Symposium on Security and Privacy*, S&P '13, 2013.

[58] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.