CFI: control flow integrity or control flow interruption?

# CFI: Control Flow Integrity or Control Flow Interruption?

Nicolò Maunero
*CINI Cybersecurity National Lab.*
Turin, Italy
nicolo.maunero@consorzio-cini.it

Paolo Prinetto
*DAUIN - Politecnico di Torino*
*CINI Cybersecurity National Lab.*
Turin, Italy
paolo.prinetto@polito.it

Gianluca Roascio
*CINI Cybersecurity National Lab.*
Turin, Italy
gianluca.roascio@consorzio-cini.it

***Abstract*—Runtime memory vulnerabilities, especially present in widely used languages as C and C++, are exploited by attackers to corrupt code pointers and hijack the execution flow of a program running on a target system to force it to behave abnormally. This is the principle of modern Code Reuse Attacks (CRAs) and of famous attack paradigms as Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP), which have defeated the previous defenses against malicious code injection such as Data Execution Prevention (DEP). Control-Flow Integrity (CFI) is a promising approach to protect against such runtime attacks. Recently, many CFI solutions have been proposed, with both hardware and software implementations. But how can a defense based on complying with a graph calculated *a priori* efficiently deal with something unpredictable as exceptions and interrupt requests? The present paper focuses on this dichotomy by analysing some of the CFI-based defenses and showing how the unexpected trigger of an interrupt and the sudden execution of an Interrupt Service Routine (ISR) can circumvent them.**

## I. Introduction

Computing devices are nowadays a corner stone of our daily life. Almost all services have now been translated into digital, and even the objects that surround us are computer-controlled and mutually connected, in what is usually referred to as the Internet of Everything. In such a scenario, ensuring security of data and privacy has become increasingly important.

Securing this new global network concerns not only the integrity and the trustworthiness of links and interconnections, but also relates to aspects strictly bound to embedded systems, such as the adopted programming languages. Most lines of code are still written in C and C++ [2], since these languages permit a good degree of low-level control without losing the advantages of high-level statements. Although, the possibility of operating at low level can turns into a disadvantage when dealing with security issues, since the direct management of memory pointers opens the door to a wide range of vulnerabilities. These include, among others, *dangling pointers* [4], i.e., pointers to live objects which are mistakenly freed and can be corrupted during the execution, and *buffer overflows* [36], i.e., out-of-bounds writes of a memory buffer which corrupts adjacent data on stack or heap.

Some of these vulnerabilities may also enable corruption of *code pointers*, used as argument of indirect control-flow transfer instructions. Malicious attackers, by tampering with them, succeed in taking full control over the program execution path. In such attacks, rather than by injecting code, a malware is executed by redirecting the flow of the program to portions of code that already exist in memory but are not meant to be executed in that order. This is the fundamental aspect of *code-reuse attacks* and famous exploit paradigms such as *Return-Oriented Programming* (ROP) [41] [12] [14] [37] and *Jump-Oriented Programming* (JOP) [9] [17]. Attackers individuate, within the code, short sequences of instructions (typically from 2 to 5) called *gadgets*. A gadget always ends with an indirect control-flow transfer instruction which can be used as a trampoline for the next gadget. By opportunely selecting gadgets and chaining them together, it is possible to force very dangerous behaviours, with Turing-complete compute capabilities [41] [46].

In [3], the enforcement of the *Control-Flow Integrity* (CFI) as basic defense was formalized. CFI dictates that, during program execution, whenever a control-flow transfer occurs, it must target a valid destination, as determined by a *Control-Flow Graph* (CFG) created at compile time. Several solutions for the CFI have been proposed [8] [22] [48] [35] [50] [30] [23] [42] [19] [21] [49]. Commonly, two phases are distinguished: during the *offline* phase, the intended flow transfers of the program are computed, and in the *online* phase it is verified whether these transfers are respected by the running program without divergencies. The offline phase is usually performed resorting to a static analysis of the program binary, finding its *basic blocks*. As in [32], a basic block is defined as a linear sequence of program instructions having one entry (the first instruction executed) and no branches out except at the exit (a control-flow transfer instruction). All statements within a basic block are executed before transferring the control to the next basic block. Transfers and basic blocks assume the identity of edges and vertices within the Control-Flow Graph, and if an online monitor (software or hardware) is able to guarantee that the program does not take paths different from those established in such a graph, then the program is considered secure and immune to redirection attacks.

The CFG represents the intended behaviour of the program, but actually can not deal with unpredictable events that may happen at runtime. In real cases, interrupt requests can be sent to the processor at any time. Serving a request is an exceptional

flow transfer, not triggered by any instruction, which preempts the execution even in the middle of a basic block, and forces the control to move to the Interrupt Service Routine (ISR) location. Such routines: (a) save the context of the current execution (i.e., the instruction pointer and the status word, as well as the used registers) on the stack, (b) acknowledge the request, (c) at their completion, they restore the context and return the control to the application. It worth pointing out that they contain normal code as any other function, including, for instance, possible local buffers that can corrupt the stack if overflowed or any other memory vulnerability. ISRs could thus be used to start a control-flow-hijacking attack, with the significant difference that, in these cases, all the static defense techniques that rely on the CFG enforcement, fail. During the offline analysis phase, both the locations from which ISRs are activated and, consequently, the return locations, are unknown. As a consequence, there is no way to monitor and protect them resorting to CFGs.

When the application runs on top of an Operating System (OS), the response to an interrupt request is demanded to the kernel. Programmers that intend to adopt a CFI enforcement solution for their programs are forced to rely on the OS capabilities to prevent possible problems related to interruptions. When instead no OS is present (bare-metal), the Interrupt Vector Table (IVT) and the ISRs are totally part of the program, so the above mentioned issues must be carefully addressed.

The present paper aims at showing how some of the classic examples of CFI enforcement, either hardware-assisted or purely software-implemented, fail in their protection purposes under the presence of hardware interrupts and vulnerable ISRs. The rest of the paper is organised as follows: Section II provides some technical background on code-reuse attacks and common solutions, while Section III analyses in details some CFI solutions and explains why and how they are vulnerable in presence of interrupts. Section IV concludes the paper.

## II. Background

In computer security, the term *Arbitrary Code Execution* (ACE) is commonly used to describe the ability to execute arbitrary commands or code on an attacked machine. ACE is achieved through tampering with the *instruction pointer* (in some architecture referred to as *Program Counter*) of a running program. The instruction pointer points to the next instruction to be executed, therefore by controlling its value an attacker can control the instruction to be executed next. To execute arbitrary code, attackers exploit possible memory vulnerabilities [36] [4] [44] present in a program to redirect the instruction pointer to malicious code, often referred to as *payload*.

Traditionally, the payload was injected together with the corrupted instruction pointer in the memory of the program (*Code Injection*) thank to vulnerabilities typically present in the stack [33]. Such exploits were made impossible after the wide adoption of *Data Execution Prevention* (DEP) [43] and *Write XOR Execute* policy [45], for which no memory location can be both writable (W) and executable (X). Attackers then reacted by devising a new attack paradigm, in which the payload is composed of code already present in the memory image of the application under attack. This was the born of the so called *Code Reuse Attacks* (CRA). The standard C library, *libc*, is the usual target, since it is loaded in nearly every program. By carefully arranging values on the stack, an attacker can cause a sequence of functions to be invoked, one after the other, with arbitrary arguments (*Return-into-libc*, [1]). DEP-based defences are thus circumvented, but still with some limitation, since fully arbitrary execution cannot be reached.

In [41], the authors stated that "*in any sufficiently large body of executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able [...] to cause the exploited program to undertake arbitrary computation*". This is the idea behind the exploit known as *Return-Oriented Programming* (ROP). ROP is based on the assumption that return addresses on the stack can point anywhere, not just to the beginning of functions. Therefore, the control flow can be hijacked through a series of small sequences of instructions, each ending with a **`ret`**, known as *gadgets*. In a large enough codebase (such as libc), there is a massive selection of gadgets to choose from, and the attackers achieve the maximum of expressiveness [46]. On the x86 platform, the attack is made stronger by the fact that, since there is no fixed instruction length, any sequence of raw bytes can be interpreted as an instruction, and the rogue return address can point even in the middle of an opcode transforming it into another.

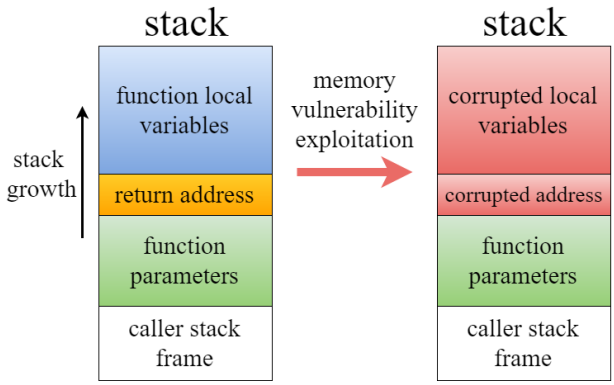


Fig. 1. Return address corruption, start point of Return-into-libc and ROP attacks.

The concept of ROP was first generalised to other architectures [12] [25] [15] [14] [31] and then extended to non-**`ret`**-ended gadgets: **`ret`** is useful in gadgets as it transfers the control flow using a program value (the return address on top of stack), not precalculated at compile time. As a result, indirect formats of **`jmp`** and **`call`** can as well be used to reach a desired instruction sequence. The concepts of *Jump-Oriented Programming* (JOP) [9] [17], *Call-Oriented Programming* (COP) [39], and others [40] [29] were introduced.

In the last years, research community and companies started elaborating and adopting different types of solution to counter

CRAs. *Address Space Layout Randomisation* (ASLR) [7] is a countermeasure taken at link-time which randomises the memory layout of the application, making it harder for an attacker to know the exact address of libraries code. Actually, in 32-bit architecture the introduced entropy is too low, and brute-force attacks can easily break the defense. Furthermore, it suffers of information disclosure, since just the base address of each segment is randomised, and therefore gaining the knowledge of a single address leads to compute the library segment base address in a straightforward manner [38].

In [20], the concept of *stack canary* or *stack cookie* was introduced: when a function is called, an additional word with a known value can be pushed on top of the stack, which is placed between the return address and the local variables. When the function returns, the value of the canary is checked, and, if it is found changed, the program is considered under attack and terminated. The canary can have a random value difficult to guess or can be composed of terminator characters, making it difficult to manipulate using input function (such as `gets()`), since terminator character breaks the input streams when recognised. However, canaries have been shown to be circumventable with more targeted stack-smashing attacks [5].

In order to address the stack smashing problem as-a-whole, a *Shadow Call Stack* (SCS) can be used [47] [26] [19] [11] [10]. Basically, at call-time, the return address is both saved on top of the normal stack and on top of an additional shadow one, accessible only by the processor in a private manner. At return time, the instruction pointer is **pop**ed from both stacks, and the values compared. If a mismatch is found, an exception is raised. Even if this solution protects the stack, it is not sufficient to fully protect an application, as it only blocks stack smashing and does not address memory vulnerability present in other segments (heap, bss, data, etc), with the consequence that exploits such as JOP can be easily performed.

*Heuristic-based* approaches claim to detect CRAs by typically monitoring the number of branches of the program and block it when suspicious behaviour is sensed. The assumption is that gadgets for ROP and JOP attacks usually consist of no more than 5 instructions. DROP [16], kBouncer [34] and ROPecker [18] are examples of heuristic-based defenders. However, it has been demonstrated that the heuristic can be easily thwarted by executing, between malicious jumps, longer sequences of non-jumping instructions or branches considered as secure [28].

Solutions presented so far can still be valid mitigation techniques, relatively simple to implement, but each of them addresses the problem of code redirection attacks just with respect to one of the vulnerabilities that lead to the exploit, without an all-encompassing vision. The paper [3] first tried to change perspective by introducing the concept of *Control-Flow Integrity* (CFI) as basic defense against CRAs, regardless of the vulnerability that may cause them. The concept behind CFI is monitoring the program at runtime to detect abnormal diversion from what is stated in its *Control-Flow Graph*. Each node in the CFG represents a *basic block*, which is a group of non-jumping instructions executed sequentially. Edges represent *branches* in the control flow, caused by jump, call, or return instructions. The CFG is defined before the execution, through a static analysis of the source code or of the binary, or by *execution profiling*, a test run which creates the possible paths. Then, at runtime, the dynamic control flow changes are restricted to the static CFG. Typically, just *indirect* formats of branching instructions are monitored, as it is usually assumed that the code is immutable, not self-modifying and not generated just-in time.

CFI policies are clustered into *coarse-grained* if the monitoring is not done by strictly enforcing the CFG, but based on simple rules, such as ensuring that **`ret`** targets are preceded by a **`call`**, or indirect calls only target prologues of functions, and similar. *Fine-grained* policies, instead, check that the execution traverses valid edges of the pre-computed CFG, only. However, coarse-grained solutions are not so different from heuristics, as both aim at distinguishing the rogue behaviours from those that *most probably* are benevolent. But *most probably* does not mean *certainly*, especially when we are dealing with clever attackers. Recent works [24] [27] show how it is possible to induce such security policies to believe that actions are within the rules when they are not. In [13] the authors showed that just 70 KB of binary code retrieved in 10 different executables within `/usr/bin` of Linux have been sufficient for mounting fully call-preceded ROP attacks.

Therefore, fine-grained CFI solutions are the only CFI policies ensuring that all control flow transfers within a program are only the ones intended by the design. In the next Section, we will compare the fine-grained CFI strategy and its most well-known implementations with the problem of unplanned interrupts, and we will show how defences can be bypassed.

## III. Main Issues of Common Solutions

### A. Control-Flow Graph

As defined in [6], the *Control-Flow Graph* (CFG) is a directed graph in which the nodes represent the basic blocks (see Section I) of a program and the edges represent the control-flow transfers. The CFG is a good instrument for outlining the behaviour of a program, but it is not sufficient to completely describe the runtime execution of a program, in particular considering *interrupts*, that can occur at any time and are thus absolutely unpredictable.

When an interrupt is triggered, a pair of "phantom" edges are created, outside the CFG: the former one connects the just-executed instruction to the initial basic block of the ISR, while the latter one connects the exit point of the ISR to the instruction following the interruption site. At offline analysis time, it is not possible to know where these "phantom" edges will be located. Moreover, their occurrence is not just untraceable, but against the definition of CFG, too, as they can start from (and arrive to) an instruction internal to a basic block.

The processor, before serving an interrupt request, saves the context of the currently executed program in order to be able to restore it when the execution is resumed. The problem is that the code of the ISR is a just another piece of

code, not immune to vulnerabilities from which an attack can start. In particular, if the routine contains one of the memory vulnerabilities presented above, the return address may be corrupted and an attacker may gain control over the program execution redirecting it to potentially dangerous code.

Given the intrinsic asynchronous nature of interrupts, no static analysis can provide a valid mean to monitor this type of transfers at runtime.

### *B. Binary Instrumentation*

The presence of interrupts is an issue not just for its non-traceability, but also for the fact that it can break the defences based on fine-grained CFI. To achieve it, several binary instrumentation approaches have been proposed.

One of these is the *label-based binary instrumentation*, first introduced by Abadi *et al.* [3] in their milestone paper. The label-based approach relies on modifying the compiled binary to insert unique IDs at the beginning of each basic block. Before each indirect branch, few instructions are inserted to check if the destination basic block's ID is in fact targeted by the instruction. Control flow tampering causes the check to fail, since the destination label ID will not match the label ID stored inside the program. Anyway, attacks are still possible if an interrupt request is served in the middle of the code used to instrument the jump.

The following code

```
cmp [ecx], 12345678h
jne violation
lea ecx, [ecx+4]
jmp ecx
```

is used to ensure that the **jmp ecx** at the bottom reaches the code starting with that ID, such as

```
.data 12345678h
mov eax, [esp+4]
...
```

Anyway, let us suppose that, thank to a memory vulnerability, an attacker has already tampered with the content of **ecx** to exploit that indirect jump. If an interrupt request is served between the **cmp** and the **jne** that triggers the violation, the processor status word (PSW) is pushed with the return address on top of the stack. The ISR may contain a vulnerability, and the PSW may be maliciously overwritten, such that the ZF flips and, when returning, the violation is bypassed and the desired piece of code is reached by the attacker.

The following instructions

```
mov eax, 12345677h
inc eax
cmp [ecx+4], eax
jne violation
jmp ecx
```

represents an alternative way to instrument the code if the destination is instrumented as follows

```
prefetchnta [12345678h]
mov eax, [esp+4]
...
```

using a side-effect-free x86 prefetch instruction. This version is even more exposed to interrupt issues: the previous exploitation is still possible if a vulnerable ISR is executed between the **cmp** and the **jne**, but another attack is possible. The **cmp** with the ID resorts to a register instead of an immediate. Therefore, let us suppose that a vulnerable ISR is triggered after the **mov** and before the **cmp**. If the ISR makes use of the **eax** register for its operations, it has to push it, and a stack corruption may permit to modify the content of **eax** with the desired label or even with the binary target of the attack. This is not unlikely: **eax** is a general-purpose register which may be used by the ISR.

An additional defense based on binary instrumentation is *Control-Flow Locking* (CFL) [8], which consists in inserting "lock" code before indirect transfer instructions and "unlock" code at each of their valid target. The lock code sets a lock variable to a value, while the unlock code, before proceeding with the execution, verifies whether the value is the lock one. The lock code also verifies if the just-executed code was unlocked and thus allowed to run, otherwise it notifies a violation. The two codes are specular:

```
L_lock: cmp lck, 0
        jne violation
        mov lck, key
        ret
        ...
L_unlock : call <function>
           cmp lck, key
           jne violation
           mov lck, 0
```

As in the label-based approach, problems may stem by the fact that the flags are possibly altered during the execution of a vulnerable ISR, so the violation can be skipped. In addition, even if the author claims that value of `lck` is stored in a protected memory, it is likely that the one seen above is not the real set of added instructions, and that `lck` is first transferred into a register. This is definitely true when this solution is to be implemented in a RISC machine, where comparisons with memory locations are not allowed. In such a situation, if an interrupt arrives during the manipulation of the lock value, and the register used is pushed because the ISR needs it, then it is possible that it may be restored as corrupted, with consequent defeat of the defense.

### *C. Hardware-assisted CFI*

CFI solutions based on code instrumentation lacks sufficient isolation of the variables and data structures that provide security, as we have shown, as well as they suffer of large overhead. Hardware-based CFI solutions try to overcome these limits. The respect of the CFG is checked at runtime via a *hardware monitor*, which in most cases is directly inserted

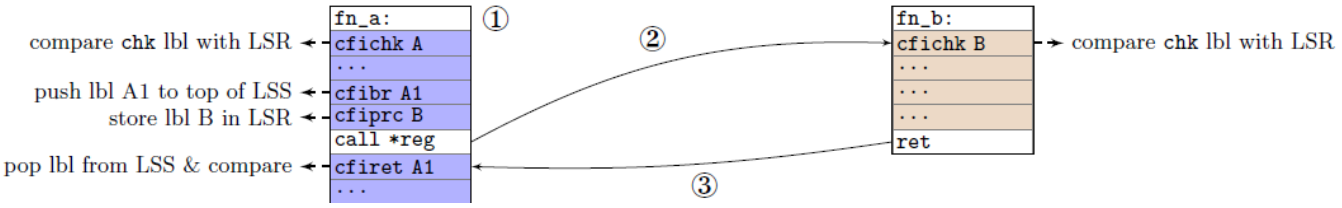


Fig. 2. Tracking of CFG in Sullivan *et al.* solution [42].

into the processor's pipeline stages [23] [42] [19] [49] but can be also attached externally to the debug interface [30].

Sullivan *et al.* [42] presented a solution in which they modified the soft processor SPARC LEON3 introducing two security-based private data structures, the "Label State Register" (LSR) and the "Label State Stack" (LSS), and five additional instructions to control them. When an indirect call is to be performed, **`cfibr`** `lbl` is executed first, which pushes a unique label on top of LSS, indicating the call site. **`cfiprc`** `lbl` then saves into the LSR a unique label for the function location. The first instruction of each function is always **`cfichk`** `lbl`, which verifies that the content of LSR is `lbl`. At return time, the processor enters in a state which only accepts **`cfiret`** `lbl`, otherwise it is blocked. So all the instructions following an indirect call are **`cfiret`** in such an architecture. Indirect jumps are instead instrumented with **`cfiprj`** `lbl`, which store `lbl` in the LSR, and at all possible destinations, **`cfichk`** `lbl` verifies that the content of LSR is in fact `lbl`.

However, the hardware implementation of the CFI enforcement does not make it immune to possible breaks due to interrupts. Referring to Figure 2, let us assume that an interrupt is triggered after **`cfibr`** `A1`. The ISR reached cannot obviously verify the caller identity with a **`cfichk`**, because it accepts a static label, unknown at code writing time. The ISR may be vulnerable, and the return address may be tampered with, and again the return site cannot be instrumented. The attacker can thus enter into the middle of any function body, and execute any wondered piece of code. At a certain time, when the **`ret`** is executed, the top of the LSS is written with `A1`, so the function returns back to the original call site, and no violation is sensed, as the **`cfiret`** performs a valid check.

## IV. Conclusions

The present paper analysed the threat of Code Reuse Attacks (CRA) and some of its countermeasures based on complying with the Control-Flow Graph (CFG) with the unpredictability of hardware interrupts. These, in fact, naturally conflict with the static nature of any pre-execution instrument and can open breaches in the defences proposed so far, independently of their actual implementation in software or in hardware. Although exploiting these vulnerabilities to successfully carry out an attack is not trivial, their presence is evident, and it is therefore advisable *to try to lock the door better before someone could learn how to open it*. Nobody can exclude that somewhere, hidden within the code, there is a sequence of even few but very dangerous instructions, such as setting a password or a key with a default value, or overwriting important memory areas, or that may be exploited to activate additional vulnerabilities to exploit later.

Fine-grained CFI solutions remain today the only effective way to defend against this type of attacks. However, especially when interrupts are frequent, such as microcontroller applications in embedded systems, this particular weakness cannot be ignored, and additional solutions must be adopted.

## V. Acknowledgments

This paper is supported in part by European Union's Horizon 2020 research and innovation programme under grant agreement No. 830892, project SPARTA.

## References

[1] The advanced return-into-lib(c) exploits: PaX case study. http://www.phrack.org/archives/issues/58/4.txt, 2001. [Online; accessed 17-June-2019].

[2] TIOBE Index of May 2019. https://www.tiobe.com/tiobe-index/, 2019. [Online; accessed 08-June-2019].

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[4] J. Afek and A. Sharabani. Dangling pointer: Smashing the pointer for fun and profit, 2007.

[5] S. Alexander. Defeating compiler-level buffer overflow protection. *The USENIX Magazine; login*, 2005.

[6] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[7] S. Bhatkar, D. DuVarney C, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, volume 12, pages 291–301, 2003.

[8] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 353–362. ACM, 2011.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[10] C. Bresch, D. Hély, A. Papadimitriou, A. Michelet-Gignoux, L. Amato, and T. Meyer. Stack redundancy to thwart return oriented programming in embedded systems. *IEEE Embedded Systems Letters*, 10(3):87–90, Sep. 2018.

[11] C. Bresch, A. Michelet, L. Amato, T. Meyer, and D. Hely. A red team blue team approach towards a secure processor design with hardware shadow stack. In *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, pages 57–62, July 2017.

[12] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[13] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, 2014.

[14] S. Checkoway, L. Davi, A. Dmitrienko, A.R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[15] S. Checkoway, A. J. Feldman, B. Kantor, J.A. Halderman, E. W. Felten, and H. Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. *EVT/WOTE*, 2009, 2009.

[16] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In A. Prakash and I. Sen Gupta, editors, *Information Systems Security*, pages 163–177, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[17] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 20–29. ACM, 2011.

[18] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attack. 2014.

[19] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis. Hcfi: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 38–49. ACM, 2016.

[20] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, , and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. 98:5–5, 01 1998.

[21] S. Das, W. Zhang, and Y. Liu. A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(11):3193–3207, Nov 2016.

[22] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.R. Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *NDSS*, volume 26, pages 27–40, 2012.

[23] L. Davi, M. Hanreich, D. Paul, A.R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. Hafix: hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference*, page 74. ACM, 2015.

[24] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, 2014.

[25] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26. ACM, 2008.

[26] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 19–26. ACM, 2009.

[27] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, May 2014.

[28] E. Göktaş, 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 *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, 2014.

[29] Y. Guo, L. Chen, and G. Shi. Function-oriented programming: A new class of code reuse attack in c applications. In *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, May 2018.

[30] Z. Guo, R. Bhakta, and I. G. Harris. Control-flow checking for intrusion detection via a real-time debug interface. In *2014 International Conference on Smart Computing Workshops*, pages 87–92, Nov 2014.

[31] T. Kornau et al. *Return oriented programming for the ARM architecture*. PhD thesis, Master's thesis, Ruhr-Universität Bochum, 2010.

[32] K. S. Kumar and D. Malathi. A novel method to find time complexity of an algorithm by using control flow graph. In *2017 International Conference on Technical Advancements in Computers and Communications (ICTACC)*, pages 66–68, April 2017.

[33] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[34] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, 2013.

[35] P. Philippaerts, Y. Younan, S. Muylle, F. Piessens, S. Lachmund, and T. Walter. Cpm: Masking code pointers to prevent code injection attacks. *ACM Transactions on Information and System Security (TISSEC)*, 16(1):1, 2013.

[36] J. Pincus and B. Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security Privacy*, 2(4):20–27, July 2004.

[37] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[38] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *2009 Annual Computer Security Applications Conference*, pages 60–69, Dec 2009.

[39] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. Pure-call oriented programming (pcop): chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques*, 14(2):139–156, May 2018.

[40] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, May 2015.

[41] H. Shacham et al. 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*, pages 552–561. New York,, 2007.

[42] D. Sullivan, O. Arias, L. Davi, P. Larsen, A. Sadeghi, and Y. Jin. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.

[43] Microsoft Support. A detailed description of the Data Execution Prevention (DEP). https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in. [Online; accessed 18-June-2019].

[44] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, May 2013.

[45] PaX Team. PaX Non-Executable Pages Design and Implementation. https://pax.grsecurity.net/docs/noexec.txt, 2003. [Online; accessed 17-June-2019].

[46] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *International Workshop on Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[47] Tzi-Cker Chiueh and Fu-Hau Hsu. Rad: a compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 409–417, April 2001.

[48] Yubin Xia, Yutao Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, June 2012.

[49] J. Zhang, B. Qi, Z. Qin, and G. Qu. Hcic: Hardware-assisted control-flow integrity checking. *IEEE Internet of Things Journal*, 6(1):458–471, Feb 2019.

[50] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, 2013.