On the Effectiveness of Control-Flow Integrity Against Modern Attack Techniques

Sarwar Sayeed(✉) and Hector Marco-Gisbert

University of the West of Scotland, Paisley, UK
{sarwar.sayeed,hector.marco}@uws.ac.uk

Abstract. Memory error vulnerabilities are still widely exploited by attackers despite the various protections developed. Attackers have adopted new strategies to successfully exploit well-known memory errors bypassing mature protection techniques such us the NX, SSP, and ASLR. Those attacks compromise the execution flow to gain control over the target successfully.

Control-flow Integrity (CFI) is a protection technique that aims to eradicate memory error exploitation by ensuring that the instruction pointer (IP) of a running program cannot be controlled by a malicious attacker. In this paper, we assess the effectiveness of 14 CFI techniques against the most popular exploitation techniques including code reuse attacks, return-to-user, return-to-libc and replay attacks.

Surveys are conducted to classify those 14 CFI techniques based on the security robustness and implementation feasibility. Our study indicates that the majority of the CFI techniques are primarily focused on restricting indirect branch instructions and cannot prevent all forms of vulnerability exploitation. Moreover, we show that the overhead and implementation requirement make some CFI techniques impractical. We conclude that the effort required to have those techniques in real systems, the high overhead, and also the partial attack coverage is discouraging the industry from adopting CFI protections.

Keywords: CFI Protection Techniques · CFI attacks

1 Introduction

Cyber Security is a changing platform, where new defense advances are being evolved every moment to cope with the ongoing challenges. Due to continuous changes in the attacking methods, a protection technique often remains outdated and having required to come up with something more advanced. From the past few decades, code-injection attack was most significant to corrupt the control-flow of a program. To meet such attacking challenges, various strong protection techniques were introduced by security developers. However, in-time the attackers have advanced their ability to corrupt control-flow in a more efficient way;

Published by Springer Nature Switzerland AG 2019
G. Dhillon et al. (Eds.): SEC 2019, IFIP AICT 562, pp. 331–344, 2019.
https://doi.org/10.1007/978-3-030-22312-0_23

hence, it was imperative to introduce another protection technique which would mitigate such leading threats.

CFI was first initiated by Microsoft in 2005 to obstruct the control-flow exploitation challenges. CFI is a security policy which can be implemented to mitigate various levels of severe attacks that mainly occur to corrupt the control-flow of a program. To accomplish an attack, an adversary goes through various attacking stages where obtaining control over the IP is the very first step of the vulnerability exploitation. A compromised control-flow may lead to various exploitation techniques, such as Code-reuse attacks (CRA), Code injection, return-to-libc. Protection techniques such as Stack Smashing Protector (SSP), Address Space Layout Randomization (ASLR) and Non-executable (NX) bit are some mechanisms that are present in all modern systems, but unfortunately, recent attacking techniques are improved to bypass all those protection mechanisms [23].

The main contributions of this paper are:

- We analyze 14 CFI techniques revealing their main features and weaknesses.
- We show the competence between hardware and software based CFI implementation.
- We conduct a survey to classify the CFI techniques based on attacks that can bypass them.
- We summarize the performance overhead revealing which of the CFI techniques are prohibitive because of the overhead introduced.

The paper is organized as follows; Sect. 2 is the background section which discusses control-flow transfers, attacks, and integrity. Section 3 defines various attack vectors that subvert the control-flow of a program. In Sect. 4, we review 14 CFI techniques and point out the limitations associated with each technique. Section 5 involves analyzing software and hardware-based CFI techniques and produces a solution by debating which techniques are more protective against control-flow attacks. In addition to that, the impact of performance overhead is also discussed. Finally the concluding Sect. 6, which summarizes the findings and discusses future work.

2 Background

To comprehend the complete control-flow mechanism this section describes types of control-flow transfers and attacks adhering to them. CFI method and its effectiveness towards control-flow attacks are also discussed.

2.1 Control Flow Transfers

Control-flow transfers can be direct or indirect. Direct control-flow transfer comprises read-only permissions; hence, these types of transfers are more secure and protection is implemented by the memory management unit (MMU). Whereas indirect control-flow transfer relies on run-time information; such as register or memory values. In a control-flow attack, attackers tend to divert the indirect control-flow to their chosen location to perform arbitrary code execution [32].

2.2 Control Flow Attacks

A control-flow attack is a run-time exploitation technique which is performed during a run-time state of a program [13]. In this attacking technique, the adversary gets hold of the instruction pointer to divert the execution flow by exploiting an application's weakness. It is also used to overwrite the buffer in the stack. Two major classes of control-flow attacks are mainly performed by random attackers; Code injection and Code-reuse attack.

2.3 Control Flow Integrity

Control-Flow Integrity is a defense policy intending to restrict unintended control flow transfers to unauthorized locations [31]. It is able to defend against various types of attacks whose primary intention is to redirect the execution flow elsewhere. Many CFI techniques [4,6,24,27–29,35,38] have been proposed over the past few years. However, they were not fully adopted due to practical challenges and significant limitations.

3 Threats

In this section, we present several attacking techniques utilized by adversaries to subvert the control-flow. The nature of exploitation strategies and their ability to perform the attacks are pointed out.

3.1 Code Reuse Attack

Code-reuse attack is an attacking technique which relies on reusing the existing code [5]. CRA exploitation occurs using codes that are already present in the target's application. For instance, the very first step of CRA begins by exploiting a vulnerability in an application that runs in the targeted system. Once the vulnerability is discovered, then the target machine can be exploited by malicious input.

3.2 Code Injection

Code injection involves injecting and executing malicious code in the memory address space [30]. The exploitation can be achieved by providing malicious payload as input and then get processed by the program. Code injection occurs when a program bug handles untrusted data. For instance, if a program does not perform bounds checking of the given input, then an adversary might provide large data than the actual limit resulting in possible buffer corruption.

3.3 Disclosure Attack

Disclosure attack endorses an attacker to uncover sensitive information, which may include source code, stack information, passwords or database information [19]. This attack can be exploited by authenticating users confidential information and then apply such information to perform further attacks.

3.4 Return-to-User

Return-to-user (ret2usr) overwrites kernel data with user address space [21]. To conduct this attack vector, an adversary gets hold of the return address, dispatch tables, and function pointers to perform arbitrary code execution. The ultimate cause involves hijacking the kernel level control-flow to redirect towards the userspace code.

3.5 Return-to-Libc

Return-to-libc occurs by jumping to the function address and allocating arguments [34]. The adversary does not require to inject payload to exploit the target. It overwrites the instruction pointer with the address pointing to the Global Offset Table (GOT), which contains pointers to glibc library functions.

3.6 Replay Attack

In this attack, an adversary copies series of data between two users and takes advantage of the event by communicating with one or both parties [25]. The adversary aims to eavesdrop the exchange of messages or aware of the message rule from earlier communications between users. Correctly encrypted message, sent by attackers, is considered as legit request and the necessary task is performed accordingly.

4 CFI Protection Techniques and Limitations

In this section, we discuss the most relevant CFI techniques that can be used to prevent control-flow hijacking. In our discussion, we also point out the limitations associated with individual techniques. Table 1 shows the enforced mechanisms in each CFI and Table 2 represents the essential characteristics related to each technique.

4.1 CFI Principles, Implementations, and Applications (CFI)

CFI was proposed by Abadi et al. [1,2] and it is the first CFI proposal for CFI implementation. They have implemented inlined CFI for windows on the x86 platform. Their work suggests that a Control flow graph (CFG) be obtained before program execution. The CFG monitors runtime behavior; therefore, any inconsistency in the program results in CFI exception to be called and application to terminate. However, Davi et al. [14] point out three main limitations of this technique. First, the source code is not always available. Second, binaries lack the required debug information and finally, it causes high execution overhead because of dynamic rewriting and run-time checks. It is also unable to determine if the function returns to the current call site [9].

4.2 CCFI: Cryptographically Enforced CFI (CCFI)

CCFI possesses new pointer arrangements, which can not be imposed with static approaches [24]. It comprises two prime attributes. First, it recategorizes function pointers at runtime to boost typecasting. Second, it restricts swapping of two valid pointers which consist of the same type. Nevertheless, CCFI comprises an average overhead of 52% on all benchmarks. CCFI is vulnerable to replay attacks [28]. It also fails to identify structure pointers. It is possible to disrupt the control flow by altering the current pointer with the old pointer. CCFI mainly focuses on defending the user level program and does not include kernel level security [24].

4.3 CFI for COTS Binaries (binCFI)

In this technique, CFI is applied to stripped binaries on x86/Linux architecture. It involves implementing CFI to the shared libraries; for instance glibc [38]. binCFI focuses on overcoming the drawbacks which are highlighted by the static analysis technique. According to Niu et al. [29], bin-CFI permits a function to return to every viable return addresses; hence, the accuracy of this CFI is fragile to Return-Oriented programming (ROP) based attacks.

4.4 Practical CFI and Randomization for Binary Executables (CCFIR)

CCFIR gathers the legit target of indirect branch instructions and places them randomly in a "Springboard Section" [37]. CCFIR restricts indirect branch instructions and permits them only towards white-list destinations. The average execution overhead of CCFIR is 3.6% and can be a maximum of 8.6%. It can be challenging to disassemble a PE file properly. CCFIR utilizes three ID's for each branch instruction, excluding the shadow stack. A ROP chain can be built to subvert CCFIR [27].

4.5 Hardware (CFI) for an IT Ecosystem (HW-CFI)

HW-CFI is a security proposal by NSA information assurance [4]. They put forward two notional features to enhance CFI. One of the features recommends implementing CFG to hardware. The other feature protects the dynamic control-flow by a protected shadow stack. This CFI proposal is a notional CFI design and might not be compatible with all system architecture. Though shadow stack can be an excellent option but monitoring shadow stack explicitly might not often be possible.

4.6 Per-Input CFI (PICFI)

PICFI imposes computed CFG to each input. It is certainly difficult to consider all inputs of an application and CFG for each of the inputs. Therefore, PICFI

Table 1. Mechanism that is enforced in CFI techniques

CFI techniques	CFI enforcement
CFI	Inlined CFI
CCFI	Dynamic Analysis
binCFI	Static Binary Rewriting
CCFIR	Binary Rewriting
HW-CFI	Landing Point
PICFI	Static Analysis
KCofi	SVA Compiler Instrumentation
Kernel CFI	Retrofitting Approach
IFCC	Dynamic Analysis
CFB	Precise Static CFI
SAFEDISPATCH	Static Analysis
C-Guard	Dynamic Instrumentation
RAP	Type Based
O-CFI	Static Rewriting

runs an application with empty CFG and lets the program to discover the CFG by itself [29]. PICFI consists overall run-time overhead as low as 3.2%. PICFI statically computes CFG to determine the edges that will be added on run-time and implements DEP to defend against code injection. However, statically computed CFG does not produce a proper result, and various experiments prove that DEP is by-passable.

4.7 KCoFI: Complete CFI for Commodity Operating System Kernels (KCoFI)

KCoFI ensures protection for commodity operating systems from attacks, such as, ret2usr, code segment modification [12]. KCoFI performs its tasks in-between the stack and the processor. KCoFI includes a conventional label-based approach to deal with indirect branch instructions. KCoFI consists of about 27% overhead on transferred file with the size between 1 KB and 8 KB and on smaller files it consists average overhead of 23%. Though KCoFI fulfills all the requirements of managing event handling based on SVA but the outcome of this CFI enforcement is too expensive, over 100% [17].

4.8 Fine-Grained CFI for Kernel Software (Kernel CFI)

Kernel CFI implements retrofitting approach entirely to FreeBSD, the MINIX microkernel system, MINIX's user-space server and partially to BitVisor hypervisor [17]. It follows two main approaches to CFI implementation. The average performance overhead ranges from 51%/25% for MINIX and 12%/17% for

Table 2. Key features of CFI techniques.

CFI techniques	Based on		Compiler modified	Shadow stack	CFG	Label	Coarse grained	Fine grained	Backward edge
	HW	SW							
CFI [1,2]		✓		✓	✓	✓	✓		✓
CCFI [24]	✓	✓	✓					✓	✓
binCFI [38]		✓				✓	✓		
CCFIR [37]		✓				✓	✓		✓
HW-CFI [4]	✓	✓	✓	✓	✓	✓		✓	✓
PICFI [29]		✓	✓		✓			✓	✓
KCoFI [12]		✓	✓			✓	✓		✓
Kernel CFI [17]		✓			✓			✓	✓
IFCC [35]		✓	✓		✓			✓	
CFB [6]		✓		✓	✓			✓	✓
SAFEDISPATCH [20]		✓	✓				✓		
C-Guard [26]		✓	✓		✓		✓		
RAP [18]		✓	✓		✓			✓	✓
O-CFI [27]	✓	✓					✓	✓	✓

FreeBSD. Though Kernel CFI cuts down the indirect transfer up to 70%; however, there still a chance remains for indirect branch instructions to transfer control to an unintended destination.

4.9 Enforcing Forward-Edge CFI in GCC & LLVM (IFCC)

Indirect Function-Call Checks (IFCC), a CFI transformation mechanism, is imposed over LLVM. IFCC introduces a dynamic tool which can be used to analyze CFI and locate forward edge CFI vulnerabilities [35]. The implementation mainly concentrates on three compiler-based mechanisms. It consists 1% to 8.7% performance overhead measured on SPECCPU2006 benchmark. Nevertheless, IFCC fails to protect against control Jujutsu attack, a fine-grained attacking technique which aims to execute malicious payload [16].

4.10 Control-Flow Bending: On the Effectiveness of CFI (CFB)

CFB comprises static CFI implementation [6]. It is based on non-control-data attacks. For instance; if arguments are overwritten directly, then that is considered as a data-only attack as it did not require to invade the control-flow for such operation, but if the overwritten data is non-control-data, then it has affected the control-flow. CFB implements fully-precise static CFG which can be undecidable [16]. CFB also violates certain functions at a high level and execution of such functions likely to alter the return address and corrupt control-flow [29].

4.11 SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks (SAFEDISPATCH)

SAFEDISPATCH defenses against vtable hijacking. It examines C++ programs statically and carries out a run-time check to ensure that the control-flow at virtual method call sites is not hijacked by attackers [20]. SAFEDISPATCH is an enhanced C++ compiler and is built based on Clang++/LLVM. The run time overhead of SAFEDISPATCH is quite low as 2.1% and having a memory overhead of 7.5%. However, all the compiler based fine-grained illustration undergoes common problems: Shared libraries; as recent programs frequently use shared library or dynamic loaded libraries [28]. SAFEDISPATCH is also unable to protect binaries; hence, making them vulnerable to ROP based attacks [27].

4.12 Control Flow Guard (C-Guard)

Control Flow Guard is a highly implemented security mechanism developed by Microsoft to defense against memory error vulnerabilities [26]. It enhances high restrictions so that arbitrary codes cannot be executed through vulnerabilities such as memory buffer overflow. However, it is unable to verify when a function returns to some unauthorized destination [18].

4.13 Reuse Attack Protector (RAP)

RAP is imposed on GCC compiler as a plugin; therefore, developers do not have to use a reformed compiler to utilize RAP [18]. RAP has a commercial version, which comes with two prime defense mechanisms to protect against control-flow attacks. However, RAP's implemented approach is very much similar to the traditional label-based approach. Label based CFI suffers from security issues as a function could return to any call site. RAP does not have a solid protection against ret2usr attacks [22].

4.14 Opaque CFI (O-CFI)

O-CFI comprises binary software randomization for CFI enforcement [27]. It protects legacy binaries without even accessing the source code. CFI checks are done on Intel x86/x64 memory protection extensions (MPX), which is hardware driven. It consists of a performance overhead of only 4.7%. O-CFI comprises static binary phase; therefore, it fails to protect codes that are generated dynamically. It is also incompatible with the Windows Component Object Model(COM). Moreover, MPX is much slower than software-based implementation and not protective against memory based errors.

5 Analysis

In this section, we assess the effectiveness of software and hardware-based CFI techniques against the threats presented in Sect. 3. The assessment involves surveying on the security experiments that are done by various research groups

Table 3. State of the art of the attacks bypassing CFI

CFI Tech.	Code reuse	Code-injection	Disclosure	Return2user	Return2libc	Replay
CFI	[7]		[7]			
CCFI						[28]
binCFI	[27]					
CCFIR			[14]			
HW CFI	[8]					
PICFI	[15]					
KCofi	[7]		[7]			
Kernel CFI	[16,33]					
IFCC	[16]					
CFB		[36]				
SafeDispatch	[14]					
C-Guard	[10]	[3]				
RAP				[22]		
O-CFI	[11]					

and then put them together to define the flaws. We establish the outcome of our evaluation by suggesting an optimal security solution and also discuss the impact of performance overhead towards CFI implementation.

5.1 Software-Based CFI

Software-based CFI enforcement primarily focuses on program instructions, which are corrupted by indirect branch instructions.

Table 3 shows that CFI [1] does not comprise strong protection and according to Chen et al. [7], it is possible to execute ROP based attacks while this technique is deployed. bin-CFI does not comprise shadow stack policy and uses ID/label for each branch instructions. Mohan et al. [27] illustrates that the most recent experiments prove that label based approaches are also vulnerable to ROP based attacks. Though CCFIR enhances a security policy that restricts indirect branch instructions to predefined functions; however, it misuses external library call dispatching policy in Linux and also causes boundless direct calls to critical functions in windows libraries which can be exploited. Moreover, the springboard section can be exploited by disclosure attacks [14]. PICFI lacks security, and the control-flow can be compromised by performing three distinguish attacking stages illustrated by [15]. KCoFI and Kernel CFI are kernel based CFI techniques. KCoFI depends only on the source code; therefore, ensuring minimal protection to binaries. It also does not provide stack protection; hence, it is exploitable by various memory error vulnerabilities, such as CRA and memory disclosure [7]. Beside that, Kernel CFI is able to build a minimal challenge to defend against ROP based attacks [33]. Table 3 also shows that IFCC is unable to mitigate control-flow exploitation and can be by-passable

by control-flow attack [16]. Though CFB enhances strong CFI enforcement by imposing shadow stack. However, it is evidenced that it can be by-passable by CFG-Aware Attack [36]. SAFEDISPATCH is a compiler based CFI enforcement focuses on securing indirect calls to virtual methods in C++, and it can also be subverted by CRA such as ROP [14]. Control Flow Guard fails to protect against indirect jumps. Moreover, It is fully by-passable by Back To The Epilogue (BATE) attack [3]. RAP makes it very hard for a ROP chain to be built up; however, it is unable to provide security against ret2usr attacks [22].

5.2 Hardware-Based CFI

Hardware-based CFI enforcement requires the system to have hardware-based components in place for deployment. Our assessment involves 3 CFI techniques, which are implemented in both hardware and software. Hardware implementation is an expensive option as it might not be compatible with the running system; hence, it may require to transform the whole system. CCFI requires AES-NI implementation besides compiler fulfillment. Experiments suggest that AES-NI can be exploited by replay attack [28]. HW-CFI comprises shadow stack to protect backward edge and landing point instructions for indirect branch transfers. In the context of security, this CFI technique will fail to mitigate indirect branch transfers if implemented. An adversary will be able to direct forward edge to any landing point instruction causing control-flow corruption [8]. Memory Protection Extention (MPX) is adopted by O-CFI; however, MPX is not a quick approach and hits 4x slow down compared to the software approach. O-CFI can also be exploited by function-reuse attacks [11].

5.3 Optimal Protection

We present that all the 14 CFI techniques comprise major limitations; hence, they are very much prone to be compromised. Table 3 does not give any indication on how much effort is required to subvert the individual CFI; however, it reveals the exploitation method, by providing a reference, related to particular CFI technique.

Based on our analysis, we are able to identify two software-based CFI techniques, which are more practical and realistic for industry deployment.

CCFIR, a coarse-grained approach, does not rely on weak implementations such as CFG or shadow stack. Since it involves binary instrumentation; hence, it does not need source code or debug information. It also protects backward edges by allowing them only towards a white-list destination. CCFIR avoids most of the weak implementation classes, which are used in most software-based CFI techniques.

RAP, a fine-grained approach, has already been adopted by the industry. A modified compiler is not required to utilize RAP. RAP enhances security by ensuring that a function is called from a designated place and returned to that

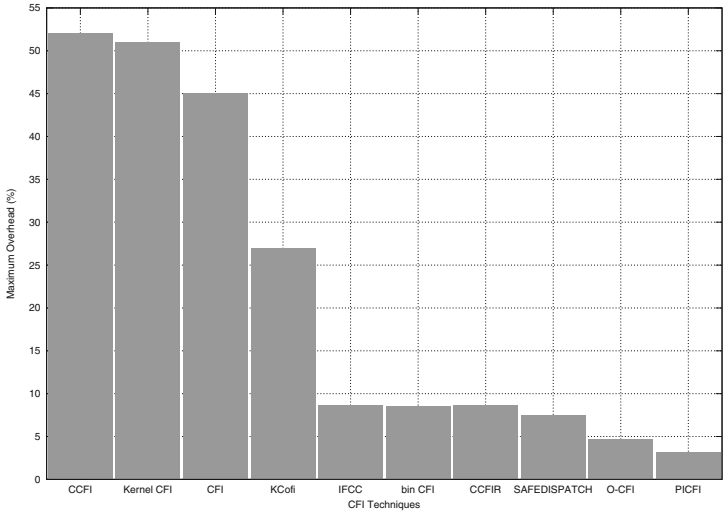

Fig. 1. Performance overhead of major CFI techniques

specific function. RAP instruments Linux kernel at compile time to implement strict CFI at runtime and assures that code pointer are not corrupted by the adversary.

Though CCFIR and RAP are not a completely secure CFI enforcement technique; however, they are able to restrict control-flow hijacking to an extensive level.

5.4 Performance Overhead

Overhead plays a significant part in CFI implementation. Figure 1 presents the maximum performance overhead of 10 CFI techniques. Distinct CFI techniques use different platforms to measure execution, performance, and space overhead. Figure 1 shows that CFI, KCofi, Kernel CFI cause the most overhead ranging from 27%–51%. CFI enforcement with such amount of overhead is not accepted and must receive a denial. bin-CFI consists considerable amount of overhead. Hardware-based enforcement, CCFI, comprises 52% overhead raising the question if hardware implementation is worth enough beside software implementation. However, O-CFI another hardware-based implementation comprises only 4.7% overhead. Compiler implemented CFI approaches, such as PICFI, IFCC, SAFEDISPATCH comprise very low overhead too. CCFIR also consists of very low overhead, 8.6%.

We evidence that an advanced CFI technique with high overhead may not be accepted since, besides integrity, performance is an important factor.

6 Conclusions and Future Work

In this paper, we have surveyed 14 major CFI techniques. It is clear that each technique comprises severe limitations and can be subverted by various attack vectors. It is identified that software-based techniques are more secure compared to hardware-based techniques and also based on practical implication. They are easy to implement and does not require an improper architectural requirement. Based on our findings, we have upheld two software-based techniques, assuming that they provide enhanced protection in-terms of security. The impact of high overheads has also been brought out in regards to CFI implementation. Our survey has established that most CFI techniques are dis-functional to provide proper security, and as a result, they were not fully acquired by the industry. Hence, future researches on CFI may consider overcoming the limitations discussed in this paper to develop a more advanced CFI implementation.

For our future work, we would like to develop a standardized method, which can be used in comparing and analyzing distinct CFI techniques so that particular information about CFI techniques and their attack types could be obtained.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, pp. 340–353. ACM, New York (2005). https://doi.org/10.1145/1102120.1102165
2. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. **13**(1), 4:1–4:40 (2009). https://doi.org/10.1145/1609956.1609960
3. Biondo, A., Conti, M., Lain, D.: Back to the epilogue: evading control flow guard via unaligned targets. In: Network and Distributed Systems Security (NDSS) Symposium 2018 (2018). https://doi.org/10.14722/ndss.2018.23318
4. NSA Information Assurance: Hardware control flow integrity CFI for an IT ecosystem. NSA, April 2015
5. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, pp. 30–40. ACM, New York (2011). https://doi.org/10.1145/1966913.1966919
6. Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R.: Control-flow bending: on the effectiveness of control-flow integrity. In: Proceedings of the 24th USENIX Conference on Security Symposium, SEC 2015, pp. 161–176. USENIX Association, Berkeley (2015). http://dl.acm.org/citation.cfm?id=2831143.2831154
7. Chen, X., Slowinska, A., Andriesse, D., Bos, H., Giuffrida, C.: StackArmor: comprehensive protection from stack-based memory error vulnerabilities for binaries. In: NDSS 2015. Internet Society, San Diego (2015). https://doi.org/10.14722/ndss.2015.23248
8. Christoulakis, N., Christou, G., Athanasopoulos, E., Ioannidis, S.: HCFI: hardware-enforced control-flow integrity. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, CODASPY 2016, pp. 38–49. ACM, New York (2016). https://doi.org/10.1145/2857705.2857722

9. de Clercq, R., Verbauwhede, I.: A survey of hardware-based control flow integrity (CFI). CoRR abs/1706.07257 (2017). http://arxiv.org/abs/1706.07257
10. Power of Community: Windows 10 Control Flow Guard Internals (2014). http://www.powerofcommunity.net/poc2014/mj0011.pdf. Accessed 15 Jan 2018
11. Crane, S.J., et al.: It's a TRaP: table randomization and protection against function-reuse attacks. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, pp. 243–255. ACM, New York (2015). https://doi.org/10.1145/2810103.2813682
12. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: complete control-flow integrity for commodity operating system kernels. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP 2014, pp. 292–307. IEEE Computer Society, Washington, DC (2014). https://doi.org/10.1109/SP.2014.26
13. Davi, L., Sadeghi, A.-R.: Building Secure Defenses Against Code-Reuse Attacks. SCS. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25546-0
14. Davi, L., Sadeghi, A.R., Lehmann, D., Monrose, F.: Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In: Proceedings of the 23rd USENIX Conference on Security Symposium, SEC 2014, pp. 401–416. USENIX Association, Berkeley (2014). http://dl.acm.org/citation.cfm?id=2671225.2671251
15. Ding, R., Qian, C., Song, C., Harris, B., Kim, T., Lee, W.: Efficient protection of path-sensitive control security. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 131–148. USENIX Association, Vancouver (2017). https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding
16. Evans, I., et al.: Control jujutsu: on the weaknesses of fine-grained control flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, pp. 901–913. ACM, New York (2015). https://doi.org/10.1145/2810103.2813646
17. Ge, X., Talele, N., Payer, M., Jaeger, T.: Fine-grained control-flow integrity for kernel software. In: Proceedings of the IEEE European Symposium on Security and Privacy, pp. 179–194, March 2016
18. grsecurity: How Does RAP Works. https://grsecurity.net/rap_faq.php. Accessed 3 Feb 2018
19. Guan, L., Lin, J., Luo, B., Jing, J., Wang, J.: Protecting private keys against memory disclosure attacks using hardware transactional memory. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP 2015, pp. 3–19. IEEE Computer Society, Washington, DC (2015). https://doi.org/10.1109/SP.2015.8
20. Jang, D., Tatlock, Z., Lerner, S.: SafeDispatch: securing C++ virtual calls from memory corruption attacks. In: NDSS 2014. Internet Society, San Diego, February 2014. http://dx.doi.org/doi-info-to-be-provided-late
21. Kemerlis, V.P., Polychronakis, M., Keromytis, A.D.: Ret2dir: rethinking kernel isolation. In: Proceedings of the 23rd USENIX Conference on Security Symposium, SEC 2014, pp. 957–972. USENIX Association, Berkeley (2014). http://dl.acm.org/citation.cfm?id=2671225.2671286
22. Li, J., Tong, X., Zhang, F., Ma, J.: Fine-CFI: fine-grained control-flow integrity for operating system kernels. IEEE Trans. Inf. Forensics Secur. **13**(6), 1535–1550 (2018). https://doi.org/10.1109/TIFS.2018.2797932
23. Marco-Gisbert, H., Ripoll, I.: On the effectiveness of NX, SSP, RenewSSP, and ASLR against stack buffer overflows. In: NCA, pp. 145–152. IEEE Computer Society (2014)

24. Mashtizadeh, A.J., Bittau, A., Boneh, D., Mazières, D.: CCFI: cryptographically enforced control flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, pp. 941–951. ACM, New York (2015). https://doi.org/10.1145/2810103.2813676
25. Microsoft: Replay Attacks (2017). https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/replay-attacks. Assessed May 2018
26. Microsoft.com: Control Flow Guard (2013). https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf. Accessed 29 Mar 2018
27. Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K.W., Franz, M.: Opaque control flow integrity. In: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, 8–11 February 2015 (2015). https://www.ndss-symposium.org/ndss2015/opaque-control-flow-integrity
28. Muench, M., Pagani, F., Shoshitaishvili, Y., Kruegel, C., Vigna, G., Balzarotti, D.: Taming transactions: towards hardware-assisted control flow integrity using transactional memory. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 24–48. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_2
29. Niu, B., Tan, G.: Per-input control-flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, pp. 914–926. ACM, New York (2015). https://doi.org/10.1145/2810103.2813644
30. OWASP: Code Injection (2013). https://www.owasp.org/index.php/Code_Injection. Accessed 28 Sept 2017
31. Pappas, V.: Defending Against Return-Oriented Programming (2015). https://www.cs.columbia.edu/~angelos/Papers/theses/vpappas_thesis.pdf. Accessed 21 Feb 2018
32. Payer, M.: Control-Flow Integrity: An Introduction (2016). https://nebelwelt.net/blog/20160913-ControlFlowIntegrity.html. Accessed 21 April 2018
33. Pomonis, M., Petsios, T., Keromytis, A.D., Polychronakis, M., Kemerlis, V.P.: kr^x: comprehensive kernel protection against just-in-time code reuse. In: EuroSys, pp. 420–436. ACM (2017)
34. Shellblade.net: Performing a ret2libc Attack (2018). https://www.shellblade.net/docs/ret2libc.pdf. Accessed 25 May 2017
35. Tice, C., et al.: Enforcing forward-edge control-flow integrity in GCC & LLVM. In: Proceedings of the 23rd USENIX Conference on Security Symposium, SEC 2014, pp. 941–955. USENIX Association, Berkeley (2014). http://dl.acm.org/citation.cfm?id=2671225.2671285
36. van der Veen, V., et al.: Practical context-sensitive CFI. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, pp. 927–940. ACM, New York (2015). https://doi.org/10.1145/2810103.2813673
37. Zhang, C., et al.: Practical control flow integrity and randomization for binary executables. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP 2013, pp. 559–573. IEEE Computer Society, Washington, DC (2013). https://doi.org/10.1109/SP.2013.44
38. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: Proceedings of the 22nd USENIX Conference on Security, SEC 2013, pp. 337–352. USENIX Association, Berkeley (2013). http://dl.acm.org/citation.cfm?id=2534766.2534796