Feel me Flow: A Review of Control-Flow Integrity Methods for User and Kernel Space

Irene Díez-Franco, Igor Santos

DeustoTech, University of Deusto
Bilbao, Spain
{irene.diez,isantos}@deusto.es

Abstract. Attackers have evolved classic *code-injection attacks*, such as those caused by buffer overflows to sophisticated Turing-complete *code-reuse* attacks. Control-Flow Integrity (CFI) is a defence mechanism to eliminate control-flow hijacking attacks caused by common memory errors. CFI relies on static analysis for the creation of a program's control-flow graph (CFG), then at runtime CFI ensures that the program follows the legitimate path. Thereby, when an attacker tries to execute malicious shellcode, CFI detects an unintended path and aborts execution. CFI heavily relies on static analysis for the accurate generation of the control-flow graph, and its security depends on how strictly the CFG is generated and enforced.
This paper reviews the CFI schemes proposed over the last ten years and assesses their security guarantees against advanced exploitation techniques.

Keywords: control-flow integrity, code-reuse attacks, operating system security

1 Introduction

Operating systems must ensure that both their own code and the code of their applications remain incorruptible and consequently secure and reliable against attackers. Since code-injection attacks are widely known, adversaries nowadays commonly exploit memory corruption bugs to subvert the *control-flow* of the operating system or the applications that are being executed within it. Rather than focusing on protecting the integrity of code, with complete memory safety or developing safe dialects of `C/C++`, modern defences try to protect the *control-flow integrity* (CFI) of these systems.

Since CFI [1] was introduced to avoid these problems and issues, different implementations and versions of this techniques have been proposed by the community that try to make it practical while ensuring the completeness of its protection. In addition, new attacks have been proposed that limit the effectiveness of these methods. Due to the raising relevance of CFI methods for the system security community, in this paper we present the first comprehensive literature review and discussion of control-flow integrity defences and the attacks that try to subvert them.

2 Control-Flow Integrity

`C/C++` code goes hand in hand with memory corruption bugs, which allow an adversary to launch attacks that exploit those memory errors. *Code injection* attacks due to stack-based or heap-based overflows, dangling pointers/use-after-free, and format string vulnerabilities are common, and can be prevented using defences such as write-xor-execute (W⊕E) / Data Execution Prevention (DEP) [4] and stack canaries [16] which are included in nowadays compilers and operating systems. Nevertheless *code-reuse* attacks, like return-into-libc [36], return-oriented programming (ROP) [44], and jump-oriented programming (JOP) [12, 5] still can not be fully prevented. Operating systems themselves are not exempt from code-reuse attacks, such as return-to-user (ret2usr) [33], a kernel level variant of return-into-libc, and sigreturn oriented programing (SROP) [6], which exploits the signal handling capabilities of UNIX like systems to deploy gadgets in the same manner that ROP and JOP do with `ret` and `jmp` instructions respectively.

Operating systems deploy statistical defences to protect user space and kernel space against code-reuse attacks; namely address space layout randomisation (ASLR) [47], and Kernel ASLR [22]. However, these defences can be circumvented due to information leakage and just-in-time code-reuse attacks both for user [45] and kernel [29] space.

Taking into account these problems, Abadi et al. introduced *control-flow integrity* (CFI) [1], a defence mechanism to prevent code-reuse attacks, which try to subvert the legitimate execution flow of a program.

CFI works in two phases, firstly, it computes the Control-Flow Graph (CFG) of the program by static analysis, either using its source code or its binary; afterwards, during program execution, CFI enforces that the program follows through the legitimate execution path; otherwise the program is aborted.

In the computation phase, CFI is concerned with *points-to analysis*, the static analysis that deals with the possible values of a pointer, because it affects the precision in which a CFG is generated [8] and consequently, the precision in which the enforcement phase will enforce the legitimate execution path. Taking into account the precision, CFI implementations can be categorised into (i) *flow-sensitive* or *flow-insensitive* and (ii) *context-sensitive* or *context-insensitive*. On the one hand, flow-sensitive algorithms use the control-flow information of a program to determine the possible values of a pointer, whereas flow-insensitive algorithms compute a set of values that are valid for all program inputs [26, 25]. On the other hand, context-sensitive algorithms take into account the context when analysing a function, preventing values from propagating to impracticable paths and thus guaranteeing that the context of a call remains independent from other call contexts; in contrast, context-insensitive algorithms allow a function to return to the computed set of all callers [52, 26].

In the enforcement phase, CFI may take into account *forward* (e.g. indirect calls or jumps) and *backward* (e.g. return instructions) control-flow transfers. CFI solutions that provide just a forward enforcement of control-flow transfers have been found insecure [23, 10, 21], whereas solutions that enforce the backward

transfers usually rely on a shadow stack, a structure that holds copies of the return addresses present in each of the stack frames of the original stack, causing up to a $\sim$ 10% increase in the program overhead [19], or use the last-branch record registers (LBR) [31, 3] which are only available to a subset of CPUs and have a limited storing capacity.

The vast majority of CFI implementations aim to protect the user space, and come in the flavours of compiler extensions, source code or binary code patching frameworks and kernel modules, whereas a small subset intend to secure the operating system deploying kernel modifications or new kernel modules.

2.1 Userland implementations

The original CFI [1] operates on x86 binaries by machine-code rewriting. For the forward control-flow transfers, the rewriting process includes a ID insertion at each destination, and a ID-check before each source; then at runtime the source ID and the destination ID must coincide. To ensure that a function call returns to the appropriate call site, a backward control-flow transfer, the implementation uses a shadow call stack relying on x86's segmentation capabilities. CFI requires (i) the code to be non-writable, to prevent attackers from rewriting the ID-check, and (ii) the data to be non-executable, to prevent attackers to execute data generated with the expected ID. The first requirement is true in modern OSes, excluding the loading time of dynamic libraries and runtime code-generation, and the second requirement is enforced with W$\oplus$E. This implementation makes the assumption that two destinations are equivalent if they are called from the same source, thus introducing imprecision in the CFG and thereby in the enforcement phase.

MoCFI [20] provides CFI protection on iOS devices' applications running on ARM processors. It addresses the special issues of ARM architecture (e.g. the nonexistence of dedicated return instructions). As the original CFI, it also operates on binaries. The authors generate a CFG of the application and a patchfile containing metadata of the indirect branches and function calls in the application; dynamic libraries used in the application are not protected. In the runtime enforcement phase, the patchfile is used by the MoCFI shared library, generating a patched application which is executed within the CFI policy. MoCFI uses a shadow stack to protect calls and returns. For the forward control-flow transfers however, it cannot protect indirect jumps/calls whose destination cannot be identified on the static analysis; thereby they can target any valid address within the function, or any valid function respectively.

Unlike MoCFI, CCFIR [55] and Bin-CFI [56] are two other binary implementations that include protection for libraries. On the one hand, CCFIR works Windows x86 PE executables, with partial support for libraries. It builds upon Abadi et al.'s approach and incorporates a third new ID-check for returns to sensitive and non-sensitive functions. This implementation suffers from the same imprecision as Abadi et al.'s for forward edges and introduces it in backward edges. On the other hand, Bin-CFI protects stripped Linux x86 binaries including shared libraries. This approach is similar to the original CFI scheme

and has lower security guarantees than CCFIR. Recent studies have found both Bin-CFI and CCFIR protections insufficient [23, 21], since grouping destinations into equivalence classes is not strong enough to prevent them for being used as ROP/JOP gadgets.

kBouncer [40] is a hardware based Windows toolkit that relies on Intel Nehalem architecture's LBR registers to retrieve the sequence of the latest 16 indirect branch instructions at critical points (e.g. system calls). In total, kBouncer protects the execution of 52 Windows API functions. Similar to kBouncer, ROPecker [14] is a Linux x86 kernel module that utilises the LBR register to prevent code-reuse attacks. Both schemes depend on chain length and gadget length heuristics to prevent such attacks. Nevertheless they can be bypassed by choosing the right sized gadget-chain length [10, 21, 24].

A recent binary based x86/64 CFI implementation, O-CFI [35], combines code randomisation with CFI checking. O-CFI first computes the permissible destination addresses for each indirect branch, then it transforms the policy that indirect branches must reach to a valid destination into a bounds-checking problem; thereby O-CFI has to check that the destination address exists within min/max address boundaries. These boundaries are protected using code randomisation and then checked making use of Intel's memory protection extensions (MPX) [30]. O-CFI uses a relaxed version of forward and backward control-flow transfer checks and consequently, can be bypassed.

All the previously presented binary level approaches [1, 20, 55, 56, 40, 35] fail to capture *complete* context sensitivity; whereas just some of them support partial (backward) context sensitivity due to the use of a shadow stack [1, 20]. PathArmor [49], is the first binary level scheme to tackle context sensitivity for forward and backward edges. Context-sensitive CFI methods need to keep track of the paths of the executed control-flow transfers, to later on enforce that the execution follows the legitimate path. Instead of using a shadow stack, PathArmor employs LBR registers to emulate a path monitoring mechanism limited by the number of LBR registers (just 16). PathArmor outperforms all previous protection schemes for forward edge transfers. However, shadow stack based approaches are still more reliable for backward edges due to the limitations that current hardware imposes.

Regarding CFI implementations that rely on source-code, Tice et al. [48] present two different forward-edge protection mechanisms integrated in production compilers, Virtual-Table Verification (VTV) and Indirect Function-Call Checks (IFCC) for GCC and LLVM respectively. Stack based attacks have been found effective bypassing VTV/IFCC [15] and the subsequent compilers have been patched. SafeDispatch [32] is a earlier LLVM compiler extension, and like VTV, aims to protect virtual tables (vtables) for `C++` virtual calls; both VTV and SafeDispatch fail to provide full control-flow protection since they focus just on forward edges. Further research has been done with the objective of protecting vtables, resulting in two binary level implementations, VfGuard [42] and VTint [54]; which unfortunately are also limited to partial control-flow protection.

A recent form of control-flow reuse attack, Counterfeit Object-oriented Programming (COOP) [43] can mount Turing-complete attacks using gadgets of `C++` virtual functions. COOP is effective against the original CFI, bin-CFI, CCFIR, VTint and partially against IFCC, VfGuard and PathArmor. In contrast, COOP can be prevented at binary level by TypeArmor [50], and at source code level with the compiler extensions SafeDispatch, VTV, VTrust [53] and VTI [7].

Niu and Tan introduced Modular CFI (MCFI) [37], a new scheme which extends CFI with modular compilation. Building upon MCFI the authors present RockJIT [38] which enforces CFI in Just-In-Time compilers; both MCFI and RockJIT induce some imprecision in the edge generation since they apply the same assumption as the original CFI for equivalent targets. Their following contribution, πCFI (per-input CFI) [39], on the contrary, introduces the highest security guarantees for a source code based CFI solution. πCFI differs from all previous CFI implementations in the way it addresses the CFG generation. Conservative CFI implementations utilise static analysis to compute the CFG before the enforcement phase, this analysis is considered hard since it has to take into account *all* the possible input values for the given program; moreover, CFI's security guarantees are strictly bounded to the CFG's precision. Niu and Tan point out than even if a perfect CFG were possible, it would still include unnecessary edges for a given input. Thereby they tackle the CFG generation in the following way; firstly, they generate the conservative CFG for all program inputs (building upon MCFI and RockJIT), then during program execution, given an input, πCFI generates CFG edges on the fly, but just those which comply with the conservative all-input CFG are enforced. This innovative scheme provides less backward edge protection compared to shadow stack approaches, but higher guarantees that other backward edge approaches. Concerning forward edges, πCFI has stronger assurance that original CFI due to the per-input mechanism.

2.2 Kernel-space implementations

State-based CFI (SBCFI) [41] is a CFI implementation for Xen and VMware Workstation virtual machine monitors. Unlike CFI enforced in userland, kernel space CFI cannot guarantee that the generated CFG is read only, nor that its data is non-executable since an attacker with access to the kernel space could also have access to page tables, and thus be able to change their properties. Thereby, SBCFI enforces a relaxed CFI by periodically checking the current kernel's CFG against the initial kernel's CFG. This implementation provides light security guarantees since it does not enforce backward edges and the support for forward edges is limited.

Hypersafe [51] is a LLVM framework extension that targets hypervisors. Hypersafe introduces the concepts of *non-bypassable memory lockdown* and *restricted pointer indexing* to introduce CFI on hypervisors. The former method is in charge of guaranteeing the integrity of the hypervisor's code and static data; the later delimits the contents of the targets of the control data (function pointers and return addresses) into a target table, to then rewrite each function

pointer or return address to a pointer index to the target table. Using the restricted pointer indexing, Hypersafe can either allow light security guarantees by allowing a function to return to any address entry on the target table, or a more strict scheme, by generating a target table for each function and allowing the function to return to a *subset* of all returns, made specifically for that function. Hypersafe implements backward edge enforcement policies but not as safe as those provided by shadow stack schemes, and for forward edges, in its strict scheme, a policy more accurate than the orignal CFI but less that the most strict user space implementations (PathArmor and πCFI).

kGuard [33] is a GCC compiler extension whose aim is to protect the kernel against ret2usr attacks. kGuard combines CFI with program shepherding. *Program shepherding* [34] is a technique that permits to implement arbitrary restrictions to code origins and control flow transfers. Upon compiling a kernel with kGuard, Control-Flow Assertions (CFA) are introduced before each control-flow transfer. These assertions are comparable to the original CFI checks, but unlike them, CFAs are not checked against a CFG to enforce a valid edge, they just ensure that the target address exists within kernel space instead. This security mechanism cannot withstand ROP/JOP like attacks since it is comparable to just enforcing weak forward control-flow transfers, like the traditional CFI, and also a weak policy for backward transfers.

KCoFI [17] provides CFI for commodity OSs utilising the infrastructure provided by the Secure Virtual Architecture (SVA) [18] virtual machine. This infrastructure is used to handle low level operations regarding the MMU, general I/O, signal dispatch and context switching. KCoFI is built on top of the SVA virtual machine, and thereby requires the OS and applications to instrument to be compiled to the virtual instruction set provided by the SVA architecture. As the original CFI, KCoFI enforces a CFI policy that is not context-sensitive.

3 Discussion

Table 1 summarises the implementations reviewed in this paper, the top part of the table lists userland implementations whereas the bottom part lists kernel space implementations. Regarding userland CFI, on the one hand, binary schemes are more common, nevertheless these schemes are known to be less secure than their source-code based counterparts. On the other hand, some source schemes (VTV, IFCC, SafeDispatch, TypeArmor) tend to focus on just forward edges, and thereby are prone to ROP attacks; while others enforce policies that fall into the *equivalent classes* paradigm which just can partially prevent ROP/JOP. Concerning kernel space, the implementations enforce modified CFI methods due to the peculiarities that protecting a kernel involves. Hypersafe is the strongest implementation followed by KCoFI. The former provides limited context-sensitivity for both edges, whereas the later falls into the equivalent classes paradigm.

In summary, the strongest implementations provide some level of context-sensitivity for both edges. PathArmor utilises the LBR registers for a hardware

Table 1. Comparison of CFI implementations. *B* stands for binary, *S* source-code, *KM* kernel module, *VMM* virtual machine monitor; *CS* context-sensitive, *EC* equivalent classes, *H* heuristics, ∅ the policy is not enforced. Regarding attacks, *Th.* stands for theoretical attacks.

		Precision		
	Scheme	**Forward**	**Backward**	**Known attacks**
Original CFI	B	EC	CS	COOP
MoCFI	B	EC	CS	Limited JOP
CCFIR	B	EC	EC	[23], COOP
Bin-CFI	B	EC	EC	[21, 23], COOP
kBouncer	B	H	H	[21, 24, 10]
ROPecker	KM	H	H	[21, 24, 10]
O-CFI	B	EC	EC	Th. ROP/JOP
PathArmor	B	Hardware limited CS	Hardware limited CS	History flush
VTV	S	CS	∅	COOP
IFCC	S	CS	∅	ROP
SafeDispatch	S	CS	∅	ROP
TypeArmor	B	EC	∅	ROP
MCFI	S	EC	EC	Th. ROP/JOP
RockJIT	S	EC	EC	Th. ROP/JOP
πCFI	S	Limited CS	Limited CS	Limited ROP/JOP
SBCFI	VMM	CFG comparison	∅	ROP
Hypersafe	S	Limited CS	Limited CS	Limited ROP/JOP
kGuard	S	Exists in kernel space	Exists in kernel space	ROP/JOP
KCoFI	S	EC	EC	Th. ROP/JOP

limited context-sensitivity, πCFI builds upon the modular CFG idea and Hypersafe uses restricted pointer indexing to provide a limited context-sensitivity.

Future trends of work are focusing on more precise context-sensitive schemes and addressing the implementation of safer CFI schemes for commodity OSs.

4 Related work

Data-Flow Integrity. Chen et al. [13] raised awareness of *non-control data* attacks the same year as Abadi et al.'s proposed CFI; one year before, Castro et al. proposed *data-flow integrity* (DFI) [11], a defence mechanism that tries to protect the legitimate data-flow analogously to CFI with control-flow.

Due to the attention that the research community has given to code injection and code-reuse attacks, attackers have directed their efforts into the creation of non-control data attacks, which have been recently found to be Turing-complete [28] and moreover, can be automatically constructed [27].

CFI mechanisms cannot withstand non-control data attacks [9] since they *follow the legitimate execution flow* and thereby pose a major threat for OSs and userland programs. The research community has proposed several approaches to

protect userland applications [11, 2] and kernel space [18, 46] but they have not been proven yet to be practical due to performance issues.

5 Conclusions

The academia has proposed several methods to protect applications and operating systems against code-reuse attacks. These approaches build upon control-flow integrity to prevent code-reuse attacks, but since control-flow integrity's effectiveness is closely bounded with the precision in which its control-flow graph is generated, not all schemes have proven to be effective against code-reuse attacks.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity: Principles, Implementations and Applications. In: CCS (2005)
2. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing memory error exploits with WIT. In: Security & Privacy (2008)
3. AMD: AMD64 Architecture Programmer's Manual Volume 2: System Programming. `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf` (2013)
4. Andersen, S., Abella, V.: Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies (2004)
5. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: CCS (2011)
6. Bosman, E., Bos, H.: Framing signals-a return to portable shellcode. In: Security & Privacy (2014)
7. Bounov, D., Kıcı, R.G., Lerner, S.: Protecting `C++` dynamic dispatch through vtable interleaving. In: NDSS (2016)
8. Burow, N., Carr, S.A., Brunthaler, S., Payer, M., Nash, J., Larsen, P., Franz, M.: Control-Flow Integrity: Precision, Security, and Performance. arXiv preprint arXiv:1602.04056 (2016)
9. Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R.: Control-flow bending: On the effectiveness of control-flow integrity. In: USENIX Security (2015)
10. Carlini, N., Wagner, D.: ROP is still dangerous: Breaking modern defenses. In: USENIX Security (2014)
11. Castro, M., Costa, M., Harris, T.: Securing software by enforcing data-flow integrity. In: OSDI (2006)
12. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: CCS (2010)
13. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-Control-Data Attacks Are Realistic Threats. In: USENIX Security (2005)
14. Cheng, Y., Zhou, Z., Miao, Y., Ding, X., Deng H., R.: ROPecker: A generic and practical approach for defending against ROP attack. In: NDSS (2014)
15. Conti, M., Crane, S., Davi, L., Franz, M., Larsen, P., Negro, M., Liebchen, C., Qunaibit, M., Sadeghi, A.R.: Losing control: On the effectiveness of control-flow integrity under stack attacks. In: CCS (2015)

16. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: USENIX Security (1998)
17. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: Complete control-flow integrity for commodity operating system kernels. In: Security & Privacy (2014)
18. Criswell, J., Lenharth, A., Dhurjati, D., Adve, V.: Secure virtual architecture: A safe execution environment for commodity operating systems. In: ACM SIGOPS Operating Systems Review (2007)
19. Dang, T.H.Y., Maniatis, P., Wagner, D.: The Performance Cost of Shadow Stacks and Stack Canaries. In: ASIACCS (2015)
20. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., Sadeghi, A.R.: MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In: NDSS (2012)
21. Davi, L., Sadeghi, A.R., Lehmann, D., Monrose, F.: Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: USENIX Security (2014)
22. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: Enhanced operating system security through efficient and fine-grained address space randomization. In: USENIX Security (2012)
23. Göktaş, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: Overcoming control-flow integrity. In: Security & Privacy (2014)
24. Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., Portokalidis, G.: Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In: USENIX Security (2014)
25. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: ACM SIGPLAN Notices (2009)
26. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (2001)
27. Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z.: Automatic generation of data-oriented exploits. In: USENIX Security (2015)
28. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In: Security & Privacy (2016)
29. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: Security & Privacy (2013)
30. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. `https://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html` (2016)
31. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. `https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html` (2016)
32. Jang, D., Tatlock, Z., Lerner, S.: SafeDispatch: Securing `C++` Virtual Calls from Memory Corruption Attacks. In: NDSS (2014)
33. Kemerlis, V.P., Portokalidis, G., Keromytis, A.D.: kGuard: lightweight kernel protection against return-to-user attacks. In: USENIX Security (2012)
34. Kiriansky, V., Bruening, D., Amarasinghe, S.P., et al.: Secure execution via program shepherding. In: USENIX Security (2002)

35. Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K.W., Franz, M.: Opaque Control-Flow Integrity. In: NDSS (2015)
36. Nergal: The advanced return-into-lib(c) exploits: Pax case study. Phrack Magazine 58 (2001)
37. Niu, B., Tan, G.: Modular control-flow integrity. In: PLDI (2014)
38. Niu, B., Tan, G.: Rockjit: Securing just-in-time compilation using modular control-flow integrity. In: CCS (2014)
39. Niu, B., Tan, G.: Per-input control-flow integrity. In: CCS (2015)
40. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent rop exploit mitigation using indirect branch tracing. In: USENIX Security (2013)
41. Petroni Jr, N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: CCS (2007)
42. Prakash, A., Hu, X., Yin, H.: vfGuard: Strict Protection for Virtual Function Calls in COTS `C++` Binaries. In: NDSS (2015)
43. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in `C++` applications. In: Security & Privacy (2015)
44. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: CCS (2007)
45. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Security & Privacy (2013)
46. Song, C., Lee, B., Lu, K., Harris, W., Kim, T., Lee, W.: Enforcing Kernel Security Invariants with Data Flow Integrity. In: NDSS (2016)
47. Team, P.: Address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt (2003)
48. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity in GCC & LLVM. In: USENIX Security (2014)
49. van der Veen, V., Andriesse, D., Göktaş, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., Giuffrida, C.: Practical context-sensitive CFI. In: CCS (2015)
50. van der Veen, V., Göktas, E., Contag, M., Pawlowski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., Giuffrida, C.: A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In: Security & Privacy (2016)
51. Wang, Z., Jiang, X.: Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: Security & Privacy (2010)
52. Wilson, R.P., Lam, S, M.: Efficient context-sensitive pointer analysis for C programs. In: PLDI (1995)
53. Zhang, C., Carr, S.A., Li, T., Ding, Y., Song, C., Payer, M., Song, D.: VTrust: Regaining Trust on Virtual Calls. In: NDSS (2016)
54. Zhang, C., Song, C., Chen, K.Z., Chen, Z., Song, D.: VTint: Protecting Virtual Function Tables' Integrity. In: NDSS (2015)
55. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: Security & Privacy (2013)
56. Zhang, M., Sekar, R.: Control Flow Integrity for COTS Binaries. In: USENIX Security (2013)