Control-flow integrity

By Contributors to Wikimedia projects

Published: 2016-06-01 · Archived: 2026-04-06 01:10:16 UTC

From Wikipedia, the free encyclopedia

Not to be confused with Common Flash Interface, the flash memory device identification standard.

Control-flow integrity (**CFI**) is a general term for computer security techniques that prevent a wide variety of malware attacks from redirecting the flow of execution (the control flow) of a program.

A computer program commonly changes its control flow to make decisions and use different parts of the code. Such transfers may be *direct*, in that the target address is written in the code itself, or *indirect*, in that the target address itself is a variable in memory or a CPU register. In a typical function call, the program performs a direct call, but returns to the caller function using the stack – an indirect *backward-edge* transfer. When a function pointer is called, such as from a virtual table, we say there is an indirect *forward-edge* transfer.[1][2]

Attackers seek to inject code into a program to make use of its privileges or to extract data from its memory space. Before executable code was commonly made read-only, an attacker could arbitrarily change the code as it is run, targeting direct transfers or even do with no transfers at all. After W^X became widespread, an attacker wants to instead redirect execution to a separate, unprotected area containing the code to be run, making use of indirect transfers: one could overwrite the virtual table for a forward-edge attack or change the call stack for a backward-edge attack (return-oriented programming). CFI is designed to protect indirect transfers from going to unintended locations.[1]

Associated techniques include code-pointer separation (CPS), code-pointer integrity (CPI), stack canaries, shadow stacks (SS), and vtable pointer verification.[3][4][5] These protections can be classified into either *coarse-grained* or *fine-grained* based on the number of targets restricted. A coarse-grained forward-edge CFI implementation, could, for example, restrict the set of indirect call targets to any function that may be indirectly called in the program, while a fine-grained one would restrict each indirect call site to functions that have the same type as the function to be called. Similarly, for a backward edge scheme protecting returns, a coarse-grained implementation would only allow the procedure to return to a function of the same type (of which there could be many, especially for common prototypes), while a fine-grained one would enforce precise return matching (so it can return only to the function that called it).

Related implementations are available in Clang (LLVM front-end),[6], GNU Compiler Collection[7], Microsoft's Control Flow Guard[8][9][10] and Return Flow Guard,[11] Google's Indirect Function-Call Checks[12] and Reuse Attack Protector (RAP).[13][14]

The LLVM compiler's C/C++ front-end Clang provides a number of "CFI" schemes that works on the forward edge by checking for errors in virtual tables and type casts. Not all of the schemes are supported on all platforms

and most of them, the exception being two "kcfi" schemes intended for low-level kernel software, depends on link-time optimization (LTO) to know what functions are supposed to be called in normal cases.[15]

Also provided is a separate "shadow call stack" (SCS) instrumentation pass that defends on the backward edge by checking for call stack modifications, available only for the aarch64 and RISC-V ISAs. And due to use of a shared processor register SCS is only enforceable on certain ABIs or if in other ways it is ensured that any other software using the register set (thread/processor) does not interfere with this use.[16]

Google has shipped Android with the Linux kernel compiled by Clang with link-time optimization (LTO) and CFI enabled since 2018.[17] Even though SCS is available for the Linux kernel as an option, and support is also available for Android's system components it is recommended only to enable it for components for which it can be ensured that no third party code is loaded.[18]

The GNU Compiler Collection implemented a "shadow call stack" compatible with Clang for aarch64 in v12 released in 2022.[19][20] This feature is primarily intended for building the Linux kernel as support is missing from GCC user space libraries.[7]

Intel Control-flow Enforcement Technology

[edit]

Intel Control-flow Enforcement Technology (CET) detects compromises to control flow integrity with a shadow stack (SS) and indirect branch tracking (IBT).[21][22]

The kernel must map a region of memory for the shadow stack not writable to user space programs except by special instructions. The shadow stack stores a copy of the return address of each CALL. On a RET, the processor checks if the return address stored in the normal stack and shadow stack are equal. If the addresses are not equal, the processor generates an INT #21 (Control Flow Protection Fault).

Indirect branch tracking detects indirect JMP or CALL instructions to unauthorized targets. It is implemented by adding a new internal state machine in the processor. The behavior of indirect JMP and CALL instructions is changed so that they switch the state machine from IDLE to WAIT_FOR_ENDBRANCH. In the WAIT_FOR_ENDBRANCH state, the next instruction to be executed is required to be the new ENDBRANCH instruction (ENDBR32 in 32-bit mode or ENDBR64 in 64-bit mode), which changes the internal state machine from WAIT_FOR_ENDBRANCH back to IDLE. Thus every authorized target of an indirect JMP or CALL must begin with ENDBRANCH. If the processor is in a WAIT_FOR_ENDBRANCH state (meaning, the previous instruction was an indirect JMP or CALL), and the next instruction is not an ENDBRANCH instruction, the processor generates an INT #21 (Control Flow Protection Fault). On processors not supporting CET indirect branch tracking, ENDBRANCH instructions are interpreted as NOPs and have no effect.

Microsoft Control Flow Guard

[edit]

Control Flow Guard (CFG) was first released for Windows 8.1 Update 3 (KB3000850) in November 2014. Developers can add CFG to their programs by adding the `/guard:cf` linker flag before program linking in Visual Studio 2015 or newer.[23]

As of Windows 10 Creators Update (Windows 10 version 1703), the Windows kernel is compiled with CFG.[24] The Windows kernel uses Hyper-V to prevent malicious kernel code from overwriting the CFG bitmap.[25]

CFG operates by creating a per-process bitmap, where a set bit indicates that the address is a valid destination. Before performing each indirect function call, the application checks if the destination address is in the bitmap. If the destination address is not in the bitmap, the program terminates.[23] This makes it more difficult for an attacker to exploit a use-after-free by replacing an object's contents and then using an indirect function call to execute a payload.[26]

Implementation details

[edit]

For all protected indirect function calls, the `_guard_check_icall` function is called, which performs the following steps:[27]

1. Convert the target address to an offset and bit number in the bitmap.
    1. The highest 3 bytes are the byte offset in the bitmap
    2. The bit offset is a 5-bit value. The first four bits are the 4th through 8th low-order bits of the address.
    3. The 5th bit of the bit offset is set to 0 if the destination address is aligned with 0x10 (last four bits are 0), and 1 if it is not.
2. Examine the target's address value in the bitmap
    1. If the target address is in the bitmap, return without an error.
    2. If the target address is not in the bitmap, terminate the program.

There are several generic techniques for bypassing CFG:

- Set the destination to code located in a non-CFG module loaded in the same process.[26][28]
- Find an indirect call that was not protected by CFG (either CALL or JMP).[26][28][29]
- Use a function call with a different number of arguments than the call is designed for, causing a stack misalignment, and code execution after the function returns (patched in Windows 10).[30]
- Use a function call with the same number of arguments, but one of pointers passed is treated as an object and writes to a pointer-based offset, allowing overwriting a return address.[31]
- Overwrite the function call used by the CFG to validate the address (patched in March 2015)[29]
- Set the CFG bitmap to all 1's, allowing all indirect function calls[29]
- Use a controlled-write primitive to overwrite an address on the stack (since the stack is not protected by CFG) [29]

Microsoft eXtended Flow Guard

[edit]

eXtended Flow Guard (XFG) has not been officially released yet, but is available in the Windows Insider preview and was publicly presented at Bluehat Shanghai in 2019.[32]

XFG extends CFG by validating function call signatures to ensure that indirect function calls are only to the subset of functions with the same signature. Function call signature validation is implemented by adding instructions to store the target function's hash in register r10 immediately prior to the indirect call and storing the calculated function hash in the memory immediately preceding the target address's code. When the indirect call is made, the XFG validation function compares the value in r10 to the target function's stored hash. [33][34]

- Buffer overflow protection

1. ^ Jump up to: ***a*** ***b*** *Payer, Mathias. "Control-Flow Integrity: An Introduction". nebelwelt.net.*
2. ^ *Burow, Nathan; Carr, Scott A.; Nash, Joseph; Larsen, Per; Franz, Michael; Brunthaler, Stefan; Payer, Mathias (31 January 2018). "Control-Flow Integrity: Precision, Security, and Performance". ACM Computing Surveys.* ***50*** *(1): 1–33. doi:10.1145/3054924.*
3. ^ *Payer, Mathias; Kuznetsov, Volodymyr. "On differences between the CFI, CPS, and CPI properties". nebelwelt.net. Retrieved 2016-06-01.*
4. ^ *"Adobe Flash Bug Discovery Leads To New Attack Mitigation Method". Dark Reading. 10 November 2015. Retrieved 2016-06-01.*
5. ^ *Endgame. "Endgame to Present at Black Hat USA 2016". www.prnewswire.com (Press release). Retrieved 2016-06-01.*
6. ^ *"Control Flow Integrity — Clang 3.9 documentation". clang.llvm.org. Retrieved 2016-06-01.*
7. ^ Jump up to: ***a*** ***b*** *"GCC Manual - Program Instrumentation Options". GCC Project. Retrieved 19 March 2026.*
8. ^ *Pauli, Darren. "Microsoft's malware mitigator refreshed, but even Redmond says it's no longer needed". The Register. Retrieved 2016-06-01.*
9. ^ *Mimoso, Michael (2015-09-22). "Bypass Developed for Microsoft Memory Protection, Control Flow Guard". Threatpost | The first stop for security news. Retrieved 2016-06-01.*
10. ^ *Smith, Ms. (23 September 2015). "DerbyCon: Former BlueHat prize winner will bypass Control Flow Guard in Windows 10". Network World. Archived from the original on September 27, 2015. Retrieved 2016-06-01.*
11. ^ *"Return Flow Guard". Tencent. 2 November 2016. Retrieved 2017-01-19.*
12. ^ *Tice, Caroline; Roeder, Tom; Collingbourne, Peter; Checkoway, Stephen; Erlingsson, Úlfar; Lozano, Luis; Pike, Geoff (2014-01-01). Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. pp. 941–955. ISBN 9781931971157.*
13. ^ *Security, heise (4 May 2016). "PaX Team stellt Schutz vor Code Reuse Exploits vor". Security (in German). Retrieved 2016-06-01.*
14. ^ *"Frequently Asked Questions About RAP". Retrieved 2016-06-01.*
15. ^ *"Control Flow Integrity — Clang 23.0.0git documentation". clang.llvm.org. Retrieved 19 March 2026.*

16. ^ *"ShadowCallStack — Clang 23.0.0git documentation"*. *clang.llvm.org. Retrieved 19 March 2026.*
17. ^ *"Clang LTO Patches Updated for the Linux Kernel - Phoronix"*.
18. ^ *"ShadowCallStack"*. *Android Open Source Project.*
19. ^ *"GCC 12 Release Series - General Improvements"*. *GCC Project. Retrieved 19 March 2026.*
20. ^ *"GCC 12 Release Series - Release History"*. *GCC Project. Retrieved 19 March 2026.*
21. ^ *"Control-flow Enforcement Technology Specification" (PDF). Intel Developer Zone. Archived from the original (PDF) on 2017-08-14. Retrieved 2021-01-05.*
22. ^ *"R.I.P ROP: CET Internals in Windows 20H1"*. *Winsider Seminars & Solutions Inc. 5 January 2020. Retrieved 2021-01-05.*
23. ^ Jump up to: ***a*** ***b*** *"Control Flow Guard"*. *MSDN. Retrieved 2017-01-19.*
24. ^ *"Analysis of the Shadow Brokers release and mitigation with Windows 10 virtualization-based security"*. *Microsoft Technet. 16 June 2017. Retrieved 2017-06-20.*
25. ^ *"Universally Bypassing CFG Through Mutability Abuse" (PDF). Alex Ionescu's Blog. Retrieved 2017-07-07.*
26. ^ Jump up to: ***a*** ***b*** ***c*** *Falcón, Francisco (2015-03-25). "Exploiting CVE-2015-0311, Part II: Bypassing Control Flow Guard on Windows 8.1 Update 3". Core Security. Retrieved 2017-01-19.*
27. ^ *"Control Flow Guard" (PDF). Trend Micro. Retrieved 2017-01-19.*
28. ^ Jump up to: ***a*** ***b*** *"Windows 10 Control Flow Guard Internals" (PDF). Power of Community. Retrieved 2017-01-19.*
29. ^ Jump up to: ***a*** ***b*** ***c*** ***d*** *"Bypass Control Flow Guard Comprehensively" (PDF). BlackHat. Retrieved 2017-01-19.*
30. ^ *"An interesting detail about Control Flow Guard"*. *Bromium. Retrieved 2017-01-19.*
31. ^ *Thomas, Sam (18 August 2016). "Object Oriented Exploitation: New techniques in Windows mitigation bypass". Slideshare. Retrieved 2017-01-19.*
32. ^ *"Advancing Windows Security"*. *Retrieved 2021-05-19.*
33. ^ *"EXTENDED FLOW GUARD UNDER THE MICROSCOPE"*. *18 May 2021. Retrieved 2021-05-19.*
34. ^ *"Exploit Development: Between a Rock and a (Xtended Flow) Guard Place: Examining XFG"*. *23 August 2020. Retrieved 2021-05-19.*
