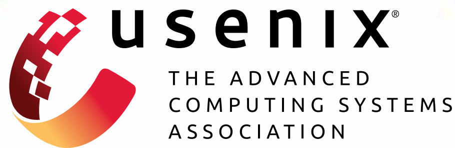


SoK: On the Effectiveness of Control-Flow Integrity in Practice

Lucas Becker and Matthias Hollick, *Technical University of Darmstadt;*
Jiska Classen, *Hasso Plattner Institute, University of Potsdam*

https://www.usenix.org/conference/woot24/presentation/becker

This paper is included in the Proceedings of the
18th USENIX WOOT Conference on Offensive Technologies.

August 12–13, 2024 • Philadelphia, PA, USA

ISBN 978-1-939133-43-4

Open access to the
Proceedings of the 18th USENIX WOOT
Conference on Offensive Technologies
is sponsored by USENIX.

SoK: On the Effectiveness of Control-Flow Integrity in Practice

Lucas Becker
Technical University of Darmstadt
lbecker@seemoo.de

Matthias Hollick
Technical University of Darmstadt
mhollick@seemoo.de

Jiska Classen
Hasso Plattner Institute, University of Potsdam
jiska.classen@hpi.de

Abstract

Complex programs written in memory-unsafe languages tend to contain memory corruption bugs. Adversaries commonly employ code-reuse attacks to exploit these bugs. Control-flow Integrity (CFI) enforcement schemes try to prevent such attacks from achieving arbitrary code execution. Developers can apply these schemes to existing code bases by setting compiler flags, requiring less effort than rewriting code in memory-safe languages. Although many works propose CFI schemes and attacks against them, they paid little attention to schemes deployed to end-users. We provide a systematic categorization and overview of actively used CFI solutions. We then conduct a large-scale binary analysis on 33 Android images of seven vendors and two Windows builds for different hardware architectures to study CFI utilization in practice. We analyzed over 77,000 files on the Android images. We found that depending on the variant, up to 94% of binaries and 93% of libraries are unprotected. All analyzed binaries depend on unprotected libraries, therefore rendering CFI enforcement ineffective. Further, we look at the development of CFI coverage over time on Android and find it stagnating. CFI roll-out is closer to complete on the Windows builds, but not all files are protected yet (2.63% unprotected). Consequently, our results show that the adoption of CFI protection is lacking, putting devices at risk. Additionally, our results highlight a large gap between the state of the art in research and the reality of deployed systems.

1 Introduction

Memory safety vulnerabilities make up two thirds of security issues in large code bases across the industry [45]. Despite the ongoing effort to prevent and mitigate memory corruption attacks, adversaries exploit these memory corruption bugs to take over computer systems. Rewriting memory-unsafe code in memory-safe languages reduces this attack surface [101]. However, the tremendous engineering effort of, e.g., porting C/C++ code to Rust, will still take years and is often infeasible on a limited budget. As a generic solution fitting most code bases, compiler toolchains add checks meant to prevent the exploitation of memory safety vulnerabilities. Control-flow Integrity (CFI) enforcement schemes are one instance of such checks. CFI checks prevent code-reuse attacks by limiting the allowed targets for indirect control-flow transfers. Ideally, this means that the program flow stays within the intended boundaries. Because the precise and sound points-to analysis required to enforce this property is generally undecidable [93], practical CFI schemes have to settle for less precise policies. Implementations must be efficient to be deployed on real-world systems while also granting sufficient security guarantees. As a result of this trade-off, coarse-grained CFI schemes can often be observed in practice, even though their ineffectiveness is well known [31]. We address the following research questions in this paper:

1. Which CFI schemes are found in practice?
2. Where and how consequently are they deployed?
3. What are their capabilities and limitations to prevent attacks?

In contrast to previous works comparing and benchmarking CFI schemes [19, 33, 65, 66, 78, 105, 114, 123], we study real-world ecosystems that deploy CFI mitigations. With this approach, we address how effectively CFI enforcement is deployed on actual systems rather than comparing academic research prototypes. For that, we study three different software- and four hardware-based CFI implementations on their corresponding platforms. We primarily focus on CFI schemes targeting user-space programs, even though most of them are used to protect the operating system kernel as well, since protecting OS kernels requires a different threat model. We also examine three shadow stack designs used to implement backwards-edge CFI. Numerous choices are involved in designing CFI enforcement schemes. These choices include which kind of control-flow transfers are protected, how the allowed Control-flow Graph (CFG) is derived, and whether special hardware features are required. CFI enforcement opens up a considerable research area, with a vast amount of different proposals [2, 24, 34, 46, 52, 59, 61, 62, 68, 72, 79, 82, 83,

87, 88, 110, 117, 118, 127]. Most of these proposals are not widely deployed in practice, as they depend on specialized hardware, require intrusive changes, come with a significant performance overhead, or are closed-source. Many promising academic solutions have not been adopted in practice and were not maintained over time. Following the approaches laid out by these prototypes, all of the most common operating systems [100] support some form of CFI enforcement in 2024. We identified the most notable solutions currently used in practice as:

- LLVM Clang CFI [107, 110], used primarily on Android and the Linux Kernel,
- Windows Control Flow Guard (WCFG) [76] and its successor eXtended Flow Guard (XFG) [120],
- ARMv8 Pointer Authentication (PA) [96] including Branch Target Identification (BTI), utilised by recent Apple Systems on a Chip (SoCs) starting with the A12, S4, and M1 chips [11], by Android, and by Windows on ARM [121]; and
- Intel Control-flow Enforcement Technology (CET) [54, 56], supported on Intel processors starting with the 11th Gen [55] and used by Windows and Linux.

There are also a few other commercial offerings, such as the Reuse Attack Protector (RAP) [49] and similar. We do not include them in this work, as it is difficult to reason about how frequently they are deployed.

We find that many binaries and libraries are missing appropriate protection, despite the compilation toolchains for these systems supporting them. On Android, we find that every investigated binary depends on at least one unprotected library. Overall, less than 17% of the binaries and libraries in recent firmware images are CFI protected. On Windows, CFI coverage is much higher, but a fine-grained CFI implementation is only available on preview builds. In summary, our main contributions are as follows:

- We systematize prevalent CFI solutions in practice, including LLVM's CFI scheme and Microsoft's closed-source implementations WCFG and XFG on Windows.
- We study CFI coverage, security characteristics, and effectiveness in practice by running a large-scale binary analysis on Android and Windows binaries.
- We analyse Android firmware releases of the same devices to get insights into the development over time.

2 CFI Design Space

Approaches to CFI Enforcement CFI schemes prevent deviation from a program's control flow, assuming an attacker who can divert the control flow by exploiting memory corruption bugs. Figure 1 shows a simplified CFG example, where basic block `A` is allowed to call blocks `B` and `C`, but not block `D`. Calling into `D` from `A` violates the CFI policy. Block `D` could, for example, be the `system()` function on Unix-like systems.

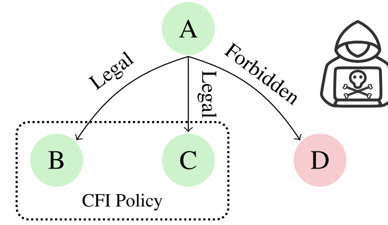


Figure 1: Simplified CFG under a CFI policy. Flows from `A` to `D` are unintended by the programmer and are only made possible by memory corruption attacks.

To protect indirect control-flow transfers, most CFI enforcement schemes follow the same basic pattern: First, a program-specific CFG is derived from the policy specifying the rules for valid control-flow transfers. Then, during runtime, this CFG is enforced by guard code, which checks that a control-flow transfer abides by the CFG [123]. If a violation of the CFG is detected, the program can be terminated to prevent successful attacks. Some recent proposals also refine the CFG during runtime [34, 52, 83, 115]. This allows to increase the precision of the CFG, for example, to achieve forms of context sensitivity. Although CFI includes forward- and backward-edge protection, this approach is often only applied to forward-edge flows, while shadow stacks are the preferred method to protect backward-edge transfers [21]. They can leverage that the return address after a call instruction is known to be the address of the subsequent instruction, language features that require special stack unwinding aside.

Compile-time Instrumentation vs. Binary Rewriting Guard code can be added directly during a program's compilation or by applying binary rewriting or instrumentation techniques. Hereby, there is a trade-off between applicability and precision: Compiler-based CFI implementations require the source code of applications to add protection, which implies that protection can only be added to commercial off-the-shelf software by the vendor itself. However, binary rewriting suffers from higher complexity and usually a loss of precision [85, 114]. Seemingly for this reason, we observed that all CFI schemes found in practice are compiler-based.

Policy Precision CFI schemes are often categorized into coarse- and fine-grained schemes. We adopt the definition from [83], wherein the number of supported *Equivalence Classes* is used as the decisive characteristic. Targets of indirect control-flow transfers are divided into classes so that if a target is reachable from a given control-flow transfer, every other target in the same equivalence class is a valid target as well, but others are not. Coarse-grained CFI schemes support only a program-independent and typically low number of equivalence classes. Fine-grained CFI schemes support a program-dependent number of equivalence classes, allowing

Table 1: Overview of CFI schemes used in practice

Scheme	Edge	Policy	Granularity	Impl.	Open-source	Platforms
LLVM CFI [107, 110]	→	Type-based	Fine	SW	✓	All LLVM supported
Control Flow Guard [76]	→	Marked function	Coarse	SW	✗	Windows
eXtended Flow Guard [120]	→	Type-based	Fine	SW	✗	Windows
Pointer Authentication [12, 96]	⇄	Implementation dep.	n/a	HW	◐	ARMv8.3-A, ARMv8.1-M
Branch Target Identification [12]	→	Label-based (#l=3)	Coarse	HW	◐	ARMv8.5-A, ARMv8.1-M
Indirect Branch Tracking [56]	→	Label-based (#l=1)	Coarse	HW	◐	Intel 11th / 12th gen.
FineIBT [44]	→	Implementation dep.	n/a	Hybrid	◐	Linux with IBT support
LLVM Shadow Call Stack [109]	←	Shadow Stack	n/a	SW	✓	ARM-based
SafeStack [64, 106]	←	Shadow Stack	n/a	SW	✓	All LLVM supported
CET Shadow Stack [56, 97]	←	Shadow Stack	n/a	HW	◐	Intel 11th / 12th gen.

"Edge" specifies the protected control-flow transfers: backward-edge (←), forward-edge (→), and both (⇄). The "Impl." column shows whether a scheme is implemented in software (SW) or hardware (HW). ◐ means that open-source implementations of the compiler and runtime components exist, but the hardware implementation is closed-source. For hardware schemes, "platforms" specifies the minimum CPU or instruction set.

each indirect control-flow transfer to have its own targets.

Evaluating CFI Effectiveness How to precisely quantify the effectiveness of CFI schemes is an open research question. To address this issue, several metrics to quantize security guarantees have been proposed, most notably Average Indirect target Reduction (AIR) [127], Average Indirect targets Allowed (AIA) [46], Relative Average Indirect target Reduction (RAIR) [117], Calltarget Reduction (CTR) [81], and Quantitative Security (QS) [19]. The common shortcoming of these metrics is that they only consider the target reduction while ignoring the *quality* of the corresponding targets. Consequently, good values in these metrics do not guarantee better security, as even with CFI, there can remain valid paths to divert the program flow maliciously. CFInsight [43] uses the length and number of such paths reaching syscalls to judge the ease of mounting attacks. We argue that this approach shares the same issue as the other metrics since it remains unclear which non-syscall gadgets are available and how path lengths correspond to exploitability.

Another approach is to collect gadgets useful to an adversary and measure their availability with and without CFI enforcement [30, 97]. In this case, it has to be defined which gadgets are considered useful. Multiple approaches exist to analyze gadget quality by determining the expressiveness of gadgets and their capabilities to set up function calls [18, 42]. We are unaware of any CFI-related work that uses such metrics for their evaluation.

3 Adversary Model and Known Attacks

CFI enforcement is a mitigation technique that aims to prevent code-reuse attacks by restricting the allowed targets of indirect control-flow transfers [2]. Therefore, CFI enforcement is intended to prevent even a strong adversary from executing arbitrary code [2, 64, 97, 110]. This adversary can read and write from/to arbitrary addresses in memory by exploiting already existing memory corruption vulnerabilities. The CFI adversary model assumes that by using these capabilities, the adversary can break Address Space Layout Randomisation (ASLR) [84, 128]. By common assumption, the adversary can perform arbitrary calculations, for example, by sending data to their server or by abusing existing scripting capabilities as present in web browsers. Since an adversary with arbitrary write capabilities could overwrite any checks, the enforcement of a Write ⊕ Execute (W⊕X) policy [104] is typically assumed to protect the integrity of code sections. Because CFI focuses on protecting individual control-flow transfers, CFI schemes generally cannot prevent data-only attacks, which only modify non-control data [21, 23].

Attacks Several generic attacks on CFI are known in the literature. The first category of attacks exploits imprecision in the enforced CFG. For instance, [31] studies Call-preceded Gadgets, assuming that the backward-edge protection only restricts returning to a legitimate call site but does not restrain the choice of call sites. This does not hold for shadow stacks, and only to some extent for PA, as discussed in Section 4.2.1, and is hence not fully applicable to programs that are adequately protected with either a hardware-based shadow call stack or PA. In the same category, [50] analyses the availability of so called Entry Point Gadgets, which are sequences of useful instructions that start at a function's entry point and end with an indirect call or jump. Similarly, [38] introduces the notion of Argument Corruptible Indirect Call Site (ACICS) gadgets, which are pairs of indirect call sites and security-sensitive target functions that are reachable from the corresponding call sites. As the name suggests, a core property of these ACICS gadgets is that the attacker can control the arguments of the corrupted call site to gain additional capabilities (e.g., arbitrary code execution in the best case).

Fundamentally, the previously covered CFI schemes can only limit the number of available gadgets, not guarantee their absence. For coarse-grained schemes such as WCFG

and Indirect Branch Tracking (IBT), this means that the set of potential entry points of ACICS gadgets consists of all functions that are marked as valid call targets. Fine-grained CFI implementations like Clang's schemes and XFG limit valid call targets per call site even more. Their protection implies that entry point gadgets must be chained so that the associated type of the call site at the end of the gadget matches the type of the next gadget or the gadget dispatching function. This exact scenario is covered by [41], which uses so-called Linker Gadgets to traverse the CFG in a policy-adhering fashion. Finally, the Counterfeit Object-oriented Programming (COOP) technique [95] chains fake objects with virtual function table pointers pointing to the functions to be called. This approach only works if C++ semantics are not adequately enforced, and hence is only applicable to WCFG, BTI or IBT, but not the type-based LLVM CFI and XFG schemes.

Besides these works, there are studies covering interactions between compilers, runtime, and CFI schemes leading to bypasses. Such interactions include the compiler spilling sensitive registers to the unprotected stack [29], compiler-introduced double-fetches that enable Time-of-check to Time-of-use (TOCTOU) attacks [122], and exception handling mechanisms that can be abused for control-flow hijacking [36]. Further works focus on data-only attacks to bypass CFI [21, 23, 58]. Such attacks break most CFI schemes since they fall outside the typical CFI adversary model.

4 CFI Scheme Internals

In this section we categorize existing schemes that we found relevant in practice and describe how they work. Refer to Table 1 for an overview.

4.1 Software-based Forward-edge CFI

CFI mechanisms for forward- and backward-edge protection can be implemented either purely in software [19, 66, 123] or based on hardware support [33, 105]. From the security perspective, we found that existing hardware-based forward-edge CFI mechanisms are not inherently more secure than schemes implemented entirely in software. Although Clerq et al. argue in [33] that software CFI instrumentation code can be bypassed if the adversary can change the page permissions of code to writable, this also applies to hardware-based schemes such as Intel's CET or ARM's PA and BTI. In addition, there is already the W$\oplus$X policy to prevent such attacks, which is typically hardware-enforced [104]. Under it, an adversary must first overcome CFI to disable this policy, at which point CFI has already been broken.

4.1.1 LLVM Clang CFI

LLVM's CFI implementation [107, 110] is part of the compiler front-end Clang and supports languages in the C family, including C++. It protects indirect function calls, calls via pointers to member functions, virtual function calls, non-virtual function calls using polymorphic classes (i.e., classes declaring or inheriting virtual functions), and invalid casts of polymorphic classes[1].

The enforced policy follows the type system of the source language, e.g., a function pointer of a specific type is only allowed to call functions with a compatible signature. Consequently, all unique function signatures and class hierarchies form their own equivalence classes, and LLVM CFI is, therefore, a fine-grained CFI scheme. LLVM's CFI checks can be divided into inlined local checks performed in the current module and Cross-Dynamic Shared Object (CDSO) checks crossing library boundaries.

Local CFI Local checks use a bit-vector-based approach. For indirect function calls involving function pointers or pointers to member functions, a jump table is generated during compilation for each unique function signature, which contains all related address-taken or exported functions. In addition, each call site is instrumented with instructions that check whether the call target is a member of the table belonging to the static type of the function pointer. Virtual and non-virtual function calls and casts to polymorphic classes are checked with a bit-vector, encoding valid vtable address points for the corresponding class type [108]. This more elaborate check is necessary because sub-classes may implement new virtual functions, resulting in vtables of different sizes, so a simple alignment and range check does no longer work.

Cross-DSO CFI When the program calls an exported function of another module, its type identifier must be derived to perform the CFI check. Hence, a direct table- or vtable- based check is infeasible in such a case, as the address of the correct table is unknown. To solve this problem, CDSO-compatible modules export the `__cfi_check` function, which is invoked by the calling module with the type identifier of the function pointer or class used in the checked call-site, and the address of the target function. This function can then check to confirm that the given target address has a matching type.

The corresponding module must be determined to find the correct `__cfi_check` function belonging to a target address. For that, CDSO-compatible programs maintain a CFI-shadow mapping that allows getting the `__cfi_check` address of the module a given address is located in. The lookup of the entry in the CFI-shadow and calling the correct `__cfi_check` function is handled by `__cfi_slowpath`. At runtime, functions affecting loaded modules such as `dlopen` must be intercepted to adjust the CFI-shadow mapping accordingly.

We find that the necessity to update the CFI-shadow mapping introduces a potential race condition, which we discuss further in Section A.2 in the appendix.

Unprotected Libraries LLVM's CDSO CFI scheme allows loading unprotected libraries (i.e., without `__cfi_check`). In

[1] We focus on control-flow transfers in this paper, as casts are not a typical concern of CFI. LLVM just uses the same mechanism to check them.

this case, the corresponding library is marked as unchecked in the shadow mapping, and indirect calls to targets in it always succeed. This principle applies even to protected indirect calls that are not intended to target functions in this library. If such calls are corrupted to transfer into an unprotected library, `__cfi_check` will dispatch them successfully, because no information is available regarding valid targets in this library. As a consequence, mixing protected and unprotected libraries diminishes CFI's security guarantees, as large libraries are bound to contain useful gadgets. Our evaluation in Section 5.1 shows that this is a serious issue across all major Android-based platforms.

4.1.2 Windows Control Flow Guard

Microsoft Windows has a proprietary CFI implementation, which is integrated into the operating system itself. It is called Control Flow Guard (WCFG) [76], and was first released in November 2014 [14]. WCFG enforces a CFI policy where indirect calls must target a known address-taken or exported function. This means there is only a single equivalence class, and WCFG is hence a coarse-grained CFI scheme. Indirect calls, including virtual calls using a vtable, are either protected with a call to a check function or entirely replaced with the call to a dispatch function that performs the WCFG check and dispatches the call afterward.

Implementation To mark functions that can be called indirectly, the `Load Configuration` structure that is part of the portable executable (PE) format is added to the executable during compilation. This structure contains various WCFG-related fields, including function pointers to the check/dispatch functions and the address of the table containing the relative addresses of all WCFG-protected functions [77]. Scanning this table when dispatching an indirect call is inefficient, which is why a bitmap marking valid functions is constructed when loading a program [124]. As the compiler aligns functions to 16-byte boundaries, a single bit per 16 bytes of address space would be sufficient to mark functions in the bitmap. Windows uses two bits to support unaligned functions, e.g., handwritten assembly.

Security Previous research identified various weaknesses in WCFG, such as gadgets that are contained in unaligned 16-byte blocks [15], or memory-based indirect calls via writable function pointers [102]. Independent of WCFG, multiple works raise issues of coarse-grained CFI schemes [31, 50, 95], implying that WCFG cannot prevent memory corruption attacks from achieving arbitrary code execution.

4.1.3 eXtended Flow Guard (XFG)

Microsoft is developing a WCFG successor called eXtended Flow Guard (XFG) [120], which is already available on Windows preview builds, even though undocumented. XFG uses a type-based policy similar to LLVM's CFI implementation (cf. Section 4.1.1), but it is based on embedded labels to perform CFI checks. We extend existing third-party works treating XFG [39, 73] by reverse-engineering relevant XFG internals to compare its security properties with the other CFI schemes. In the implementation at the time of writing (Insider Preview build 23440), a 64-bit type hash precedes all XFG instrumented functions. During runtime, the XFG dispatch function checks whether a given call target has the expected type hash or else the program is terminated. The type hashes are derived from a combination of a function's signature, its name, and the class hierarchy in case of virtual function calls. Consequently, cross-module calls to XFG-instrumented functions work without additional overhead since type hashes directly precede the functions. On some architectures such as x64, `MOV` instructions for loading the expected type hash contain the type hash itself as part of the instruction encoding. Such instructions would then produce unintended call targets. To address this issue, the instrumentation code loads the expected type hash with the last bit flipped, and the dispatch function undoes this bit flip before comparing it with the stored label. Figure 2 depicts the whole XFG flow: First, an instrumented call site is redirected to the dispatch function. This function is configured by the loader, which sets the `__guard_dispatch_icall_fptr` function pointer depending on whether WCFG or XFG should be enforced. The XFG dispatch function loads the hash located at the quad-word prior to the target address, flips the last bit of the expected value, and compares them. If they match, the call is dispatched. Else, the WCFG bitmap is consulted to check if the target is a known function entry address. The target address is called if it is. Otherwise, a function is called to determine the consequences of this CFI violation.

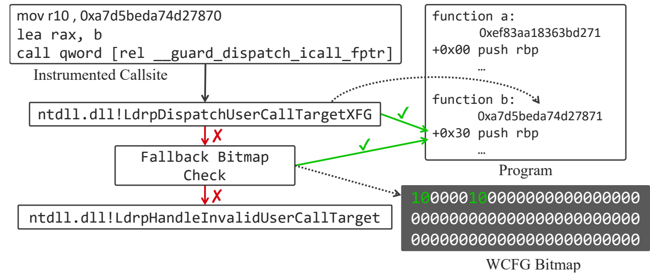


Figure 2: Reverse-engineered XFG check flow. Green and red arrows represent flows after a successful and failed check, respectively. The dotted arrows mark data fetches.

XFG is backward-compatible with WCFG-protected programs. After a failed check for a matching type hash, the XFG dispatch function also consults the WCFG bitmap to check whether the target is a WCFG-protected function, and if so, may still allow the call. XFG-instrumented functions hence use the fourth remaining bitmap state to encode that they should not be valid WCFG call targets.

4.2 Hardware-based Forward-edge CFI

This section introduces four hardware-based CFI schemes targeting forward-edge protection. These schemes are coarse-grained, except for PA and FineIBT, which allow to implement different policies. It follows that they are less precise than LLVM CFI or XFG. However, due to their implementation in hardware, they are more efficient.

4.2.1 ARM Pointer Authentication

Pointer Authentication (PA) [12] is a security extension for the AArch64 architecture, which allows for protecting pointer integrity by inserting a cryptographic Message Authentication Code (MAC) called Pointer Authentication Code (PAC) into the unused upper bits of pointers. Unused bits are available because the virtual address space size does not occupy the full 64-bit of register width. Consequently, their exact number depends on the specific implementation. PA was introduced in `ARMv8.3-A` in 2016 [17], and later also for the microprocessor profile starting with the `ARMv8.1-M` architecture update, as announced in 2021 [80]. To operate on PACs, the PA extension adds a variety of instructions that can be divided into four categories [12]:

- `PAC*` instructions to generate and insert a PAC,
- `AUT*` instructions to authenticate and remove the PAC for subsequent use of a pointer,
- `XPAC*` instructions to strip the PAC from a pointer without authenticating it, and
- *(no common prefix)* combined instructions that perform a PA-related operation and a related instruction together.

The MAC algorithm uses one of five keys and a 64-bit context value that allows tying pointers to a specific context. These keys are stored in CPU registers and are not accessible from exception level EL0 (user space).

Since PA is more of a building block for CFI schemes rather than a mitigation on its own, there are different PA-based implementations that differ in their respective characteristics. While the protection of forward-edge flows is covered in multiple research works such as [40, 57, 68, 94, 96, 126], Apple's `arm64e` ABI [10] is the only case where we observed a PA-based forward-edge scheme in practice.

4.2.2 ARM Branch Target Identification

ARM's BTI feature is a forward-edge CFI scheme and an alternative to custom PA-based schemes. It introduces the `BTI` instruction, which takes a target operand specifying what kind of control-flow transfer is allowed to target the instruction. The target operand can be `c`, `j`, or `jc`, indicating that the corresponding `BTI` instruction can be targeted by calls, jumps, or both respectively [12]. Jumps that target the registers `X16` or `X17` are also compatible with the `c` target. This enables the use of jumps to these registers in Procedure Linkage Table (PLT) entries or for indirect tail-calls [92]. BTI allows configuring which memory page should be protected. Outside protected memory regions, the `BTI` executes as NOP [12].

4.2.3 Intel Indirect Branch Tracking

The IBT feature is the forward-edge control-flow transfer protection component of Intel CET. It is a coarse-grained CFI scheme using label instructions for marking valid call targets, and thus very similar to the proposal in the seminal work on CFI [2] and ARM's BTI feature. The two label instructions that IBT adds are `ENDBR32` and `ENDBR64`, for the 32-bit compatibility mode and the 64-bit mode, respectively.

The CFI policy enforced by IBT is straightforward: If an indirect call or jump is encountered, the next instruction executed must be a label instruction. If it is not, the control protection exception is raised [56]. There might be instances, such as switch-case constructs, where the control-flow transfer target resides in read-only memory or where IBT is undesired for some other reason. To support such instances, CET supports a no-track prefix that marks the subsequent `CALL` or `JMP` as not requiring a `ENDBR` instruction as the target. For backward compatibility, it is also possible to set up a bitmap that marks memory pages where the same exception applies [56].

4.2.4 FineIBT

FineIBT [44] is a hybrid CFI scheme, which improves the precision of coarse-grained hardware-based schemes while preserving their performance gains. While the general approach is mostly architecture-agnostic, their implementation targets Intel's BTI as suggested by the name of their scheme. The fundamental idea is that if a coarse-grained scheme like BTI or IBT protects a program, all indirect control-flow transfers are already limited to target particular instructions (i.e., `ENDBR64` for IBT), and instrumentation code only needs to be placed at these locations. This restriction means that the policy check can be executed *after* the control-flow transfer occurred since the hardware-based scheme guarantees that only such locations can be indirect call targets. Compared to full-software implementations of this approach like XFG, FineIBT avoids loading a label from memory before taking an indirect control-flow transfer. It follows that FineIBT is compatible with execute-only memory.

4.3 Software-based Backward-edge CFI

Shadow call stacks are a common approach to protect backward-edge control-flow transfers. They protect saved return addresses against memory corruption attacks by saving them to an isolated memory region. Since such metadata must be dynamically updated during runtime, it cannot be

protected by marking it read-only like CFI checks [20]. Solving this issue in software is challenging, as it either requires full Software Fault Isolation (SFI) or hardware-supported isolation [2, 20]. Often, software-based shadow call stacks rely on information hiding to protect the shadow stack area. However, it has been shown that due to information disclosure attacks [48, 84], this approach cannot withstand the CFI adversary [128]. Recent works propose re-randomization as a solution to such attacks [119, 129, 130]. They continuously re-randomize the addresses of protected areas or the addresses contained therein and thus limit the use of information leaks to an adversary. Another problem of software-based shadow stacks is that on some platforms such as x64, where call instructions directly push the return address on the stack, there is a timing window for a race condition between the call instruction and the return address being written to the shadow stack [2]. These shortcomings aside, a few software-based shadow stack approaches are found in practice.

4.3.1 LLVM Shadow Call Stack

LLVM implements a shadow stack scheme for the AArch64 architecture. The design supported by Clang is a compact shadow stack based on information hiding, i.e., the shadow stack maintains its own stack pointer in register `X18`, which must remain unknown to the attacker to offer any protection. This implies that this register must not be spilled onto the stack (e.g., when calling into unprotected code), or else an attacker could obtain the shadow stack location from there. Because of its nature as software-based shadow stack, which is only protected through information hiding, this design is inherently weak against attacks that try to uncover hidden locations in memory. Corresponding example attacks based on allocation oracles are proposed in [84]. A larger guard region can be allocated to increase the resistance against such attacks, containing the shadow stack itself. Thus, an allocation oracle will only find the whole guard region instead of the exact location of the shadow stack. Android implements this approach in its Libc [89].

4.3.2 SafeStack

Besides the shadow call stack, LLVM also implements SafeStack [64, 106]. The key idea of SafeStack is to separate safe and unsafe memory objects on the stack. Memory objects considered safe are return addresses, stack spills, and local variables that are not address-taken but only accessed via the frame pointer. Everything else is stored on the unsafe stack. The implementation relies on information hiding to protect the safe stack area and hence suffers the same associated weaknesses as the shadow call stack [47].

4.4 Hardware-based Backward-edge CFI

Hardware-based backward-edge CFI schemes address the weaknesses of software-based designs. They can implement atomic instructions to prevent race conditions and provide memory isolation for sensitive regions.

4.4.1 PA-based Approaches

One common scheme uses PA to protect saved return addresses on the stack by tying them to the stack pointer value at function entry [96]. This can efficiently be done by using the `PACIASP` and `AUTIASP` instruction pair, which sign and verify the link register with the current stack pointer value as context. Since the link register is used to store return addresses by `BL` instructions, these instructions can be placed at the start of the function prologue and epilogue, respectively. The `PACIASP` and `PACIBSP` instructions have implicit BTI behaviour, making them valid call targets [12] under BTI enforcement. Consequently, programs using PA to protect the return address with these instructions do not need an extra `BTI` instruction at the start of a function.

In comparison to a regular shadow call stack there is no memory overhead for the shadow call stack area. Neither loader nor operating system needs to do additional work besides the operating system managing the PA keys themselves, which is required for any PA-based scheme. Due to the nature of stack-based function calls, stack pointer values are not guaranteed to be unique to a specific function during program execution. This enables substitution attacks, where an adversary exchanges the saved return address with another unintended target that has been leaked earlier [96]. Consequently, compared to the hardware-based shadow call stack, the PA design offers weaker security guarantees.

4.4.2 Intel CET Shadow Call Stack

Intel CET features a hardware-based shadow call stack for backward-edge protection [97]. This shadow stack is implemented as a descending second stack designated for storing only return addresses. A new `SSP` CPU register holds the current shadow stack pointer. This register can only be modified by dedicated new instructions for shadow call stack management, which are intended for either the operating system or libraries that need to handle special stack unwinding cases. During normal program execution, the call and return instructions are shadow stack aware if the shadow stack feature is enabled. This means that the call instructions do not only push the return address to the unprotected program stack but also to the shadow stack. Similarly, the return instructions compare the return address stored on the shadow stack to the return address stored on the save stack and only continue if they match. Otherwise, the #CP exception is raised [56]. Adapting the semantics of these instructions means that existing programs do not need to be recompiled to benefit from shadow stack

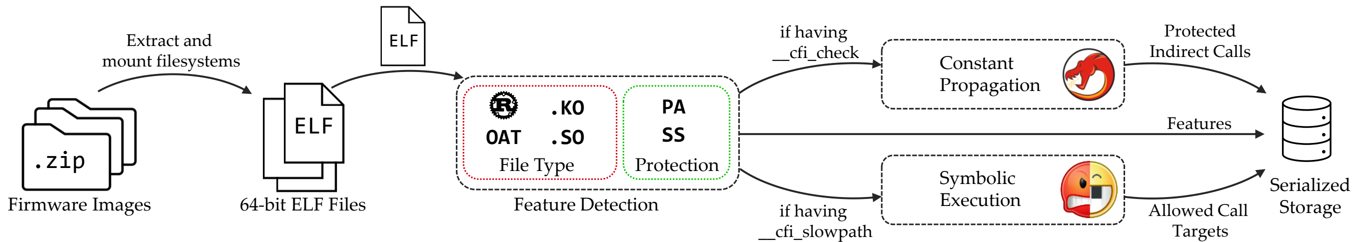


Figure 3: Android analysis pipeline.

protection as long as they do not implement custom unwinding logic. Only the runtime and standard libraries that handle unwinding must be modified to be shadow stack aware.

The shadow stack region is protected by adding a new attribute to the address translation to indicate a shadow-stack page. Pages marked as such cannot be modified by regular store instructions, protecting the integrity of the saved return addresses. In addition, call and return instructions fault if the page where they try to store or fetch the saved return address from the shadow stack, respectively, is not marked as shadow page [56]. This implies that even with complete control over a program's virtual memory, an adversary cannot manipulate return addresses without access to either gadgets with management instructions or a primitive to create shadow-stack pages under their control.

5 Study on CFI Adoption in Platforms

In this section, we study the usage and effectiveness of the CFI schemes covered in the previous section on the Android, Linux, and Windows platforms.

5.1 Android Study

The Android Open Source Project (AOSP) [7] uses LLVM's CFI implementation to enforce CFI since Android 8.1 for a set of components [4]. In user space, CDSO mode is used, enabling protected programs to perform indirect calls into shared libraries [108]. Since LLVM's CFI implementation only protects indirect forward-edge control-flow transfers, Android (on AArch64) supports LLVM's shadow call stack for backward-edge protection [6]. However, this shadow call stack is based on information hiding and is not designed to resist the typical CFI adversary. We also observed the PA-based return address protection scheme used as a more secure alternative. Bionic provides the necessary runtime support for the shadow call stack and LLVM CFI. Android 12 added support for ARM's Memory Tagging Extension (MTE) [1, 8], a hardware-based memory safety mitigation which implements memory tagging. While the first phones supporting MTE have been released [16], it is not a CFI mitigation and disabled by default, and we do not include it in our study.

5.1.1 Android Image Analysis Setup

We analyze 33 Android images of popular flagship devices to compare the CFI usage on Android across multiple device manufacturers. Based on their market share, we select the smartphones Samsung Galaxy S22, Xiaomi 13, Vivo V25, and Oppo Reno 8 5G [9]. In addition, we include the pure AOSP Generic System Images (GSIs) for Android 10 to 14 and a system image from Google's Pixel 7 phone, because Google is part of the driving force behind Android. We also include the GrapheneOS firmware for the Pixel 7, which promises increased security and privacy [27], to see if it has better CFI coverage than the Google Pixel 7 firmware. Based on the results of these images, we picked the Samsung Galaxy S20 and the Xiaomi Mi 10 for analysis over time. They both have publicly available firmware archives ranging from Android 10 to Android 13. The full versions and source URLs of all firmware images are specified in the appendix (Table 8).

Since all of these phones are ARM-based, we enumerate AArch64 ELF files on their Android firmware images and run an analysis on them. An overview of our analysis pipeline is shown in Figure 3. We extract the following characteristics:

ELF Type We distinguish between binaries (i.e., executable programs), shared libraries, and loadable kernel modules (with a `.ko` extension).

General LLVM CFI Usage To determine general LLVM usage, the analysis checks for the existence of the exported `__cfi_check` function, which is always present if the binary was compiled with LLVM CFI. We manually check samples to confirm that all vendors built their applications with CDSO.

Shadow Stack Usage The analysis searches for instructions unique to shadow-stack-protected binaries to detect shadow-stack usage. One such instruction is `ldr LR, [x18, #-8]!`, which is used to restore the link register from the shadow stack. We examine multiple random samples of files containing this instruction and found that it is only used for the shadow stack and does not appear in other contexts.

Pointer Authentication PA is used to protect return addresses in some of the files. We detected PA protection by scanning for related instructions. This approach introduces imprecision as it counts files where only certain functions are PA-instrumented as PA-protected. However, we deem this approach sufficient to understand the overall distribution of fully unprotected binaries. We also scanned for BTI related

Table 2: CFI coverage of Android firmware images.

	Files Total [count]			LLVM CFI Protected [%]				Shadow Stack Protected [%]			Pointer Auth. Protected [%]		
Vendor & Devices	Binaries	Libraries	Kernel Modules	Binaries	Libraries	Kernel Modules	Kernel	Binaries	Libraries	Kernel Modules	Binaries	Libraries	Kernel Modules
GSI 10	142	1240	0	4.93	9.19	n/a	n/a	0	0.24	n/a	0	0	n/a
GSI 11	154	1383	0	7.79	10.12	n/a	n/a	0	0.14	n/a	0	0	n/a
GSI 12	167	1708	0	8.98	10.6	n/a	n/a	0	0.18	n/a	12.57	5.62	n/a
GSI 13	173	1893	0	8.67	9.77	n/a	n/a	0	0.11	n/a	35.26	18.17	n/a
GSI 14	150	1729	0	9.33	8.04	n/a	n/a	0	0.12	n/a	39.33	22.15	n/a
Xiaomi 13	535	2601	270	47.85	65.24	100.0	✓	0	0.08	100.0	94.21	92.62	97.78
Google Pixel 7	234	998	241	7.26	16.83	100.0	✓	0	0.2	99.59	12.82	9.52	99.17
GrapheneOS Pixel 7	233	994	241	7.3	16.9	100.0	✓	0	0.2	99.59	12.88	9.56	99.17
Oppo Reno 8 5G	351	2316	9	6.27	8.25	0	✗	0	0.17	0	7.98	4.49	0
Samsung Galaxy S22	289	1775	6	7.27	12.23	100.0	✓	0	0.28	100.0	23.53	17.18	100.0
Vivo V25	361	2771	10	5.82	6.75	0	✗	0	0.14	0	11.36	4.19	0
S20 2020-02-19 10	255	1573	1	3.14	6.74	0	✗	0	0.25	0	0	0	0
S20 2020-05-15 10	254	1572	1	3.15	6.74	0	✗	0	0.25	0	0	0	0
S20 2020-10-14 10	257	1595	1	3.11	5.58	0	✗	0	0.25	0	0	0	0
S20 2020-11-23 11	264	1395	1	4.92	9.82	0	✗	0	0.22	0	0	0	0
S20 2021-05-17 11	271	1430	1	4.8	9.58	0	✗	0	0.21	0	0	0	0
S20 2021-10-20 11	274	1447	1	4.74	9.47	0	✗	0	0.21	0	0	0	0
S20 2021-12-23 12	273	1527	0	5.86	11.92	n/a	✗	0	0.2	n/a	5.86	4.32	n/a
S20 2022-04-26 12	276	1536	0	5.8	11.78	n/a	✗	0	0.2	n/a	6.16	4.62	n/a
S20 2022-09-27 12	276	1536	0	5.8	11.78	n/a	✗	0	0.2	n/a	6.16	4.62	n/a
S20 2022-10-24 13	278	1582	0	5.76	12.2	n/a	✗	0	0.19	n/a	20.14	15.49	n/a
S20 2023-02-20 13	279	1592	0	5.73	12.12	n/a	✗	0	0.19	n/a	20.07	15.39	n/a
S20 2023-07-26 13	280	1599	0	5.71	12.45	n/a	✗	0	0.19	n/a	20.0	15.82	n/a

For the S20 firmware images, the security patch level and the Android version are given. Data for the Mi 10 in Table 5 in the appendix.

instructions, but found them too rare for consideration.

Kernel CFI Configuration Clang's kernel CFI is enabled by the `CONFIG_CFI_CLANG` Kconfig flag [5]. To determine if the kernel of a firmware image enables CFI we use the `extract-ikconfig` script from the Linux repository [70] and double check the decompressed kernel image for CFI-related symbols and strings.

Rust Source Language All analyzed Rust binaries were compiled without CFI protection and hence excluded from the statistics. These files were detected by checking their symbols for Rust-specific functions (e.g., `__rust_alloc`).

OAT Files Some ELF files can contain ahead-of-time compiled DEX code in a custom OAT format [91]. Such files are not CFI-protected and can be detected by a combination of specific symbols, such as `oatdata` and `oatdex`. We exclude them from the statistics.

Library Dependencies Unprotected dependencies massively contribute to the number of available call targets. We collected each analyzed file's dependencies as indicated in the corresponding ELF structure.

Type Identifiers Passed to __cfi_slowpath Extracting type identifiers passed to `__cfi_slowpath` shows which function or class types are actually used for CFI checks. In our analysis, this is done by leveraging Ghidra's [3] constant propagation analysis. First, the analysis searches calls to `__cfi_slowpath`, which is imported from bionic and hence easy to locate. Then, it extracts the first argument that is passed to the function. Since type identifiers are constants, this is well-doable by static analysis.

Type Identifiers and Their Associated Address Ranges Given a binary file, its equivalence classes and their members can be constructed by extracting type identifiers and their associated address ranges. The core observation is that this information is encoded in the exported `__cfi_check` function, which is easy to locate. In addition, this function is independent of any global state and uses only arithmetic and control-flow instructions, making it a good fit for symbolic execution [13]. Our implementation uses the Angr framework [99]. It progresses execution states until they either hit a return, indicating a successful check, or a call to `abort()`, in which case they are discarded. Afterward, the collected constraints on the type identifier and the address argument are solved for the successful states, resulting in a mapping from type identifiers to the address ranges of their jump tables or vtables. Finally, it detects whether a target range is for a jump table or a vtable by checking whether a branch instruction is found at the start of the first slot. On a side note, the results of the symbolic execution can also support program analysis. We discuss this further in Section A.1 in the appendix.

5.1.2 Evaluation

Unprotected Binaries We count the number of files with and without CFI protection (as indicated by the existence of the `__cfi_check` export) to determine the general coverage of CFI protection. We divide them into binaries, libraries, and loadable kernel modules and additionally count the shadow call stack usage. We filter out files that are unprotected by design, such as OAT files or binaries that were classified as Rust binaries. Table 2 depicts the numerical results for the CFI coverage of the remaining files. For both binaries and libraries, only the minority of files is CFI-protected, with the

Table 3: Unprotected library dependencies in binaries.

Vendor & Device	Min [%]	Mean [%]	Max [%]
GSI 10	83.78	92.84	100.0
GSI 11	75.24	86.93	100.0
GSI 12	61.76	83.56	100.0
GSI 13	58.97	83.12	100.0
GSI 14	63.41	82.46	100.0
Xiaomi 13	24.14	75.83	100.0
Google Pixel 7	58.97	84.65	100.0
GrapheneOS Pixel 7	58.97	84.65	100.0
Oppo Reno 8 5G	61.76	87.0	100.0
Samsung Galaxy S22	60.98	87.13	100.0
Vivo V25	58.97	88.48	100.0

only exception being the Xiaomi 13 firmware. Concerning loadable kernel modules, build settings seem to be more consistently activating CFI, i.e., all kernel modules are protected, or none are.

Unprotected Dependencies When looking at the few protected binaries, an essential factor is the number of unprotected dependencies they load. Unprotected dependencies have twofold implications: First, CFI-protected control-flow transfers targeting address in them cannot be checked. Hence, every byte in all executable sections in any unprotected dependency is a valid call target. Second, indirect control-flow transfers within the unprotected dependencies are not checked and can be used to reach arbitrary code in the protected parts.

For each of the selected Android firmware images, we compute the recursive dependencies of all protected binaries. Then, we calculate the ratio of unprotected to protected dependencies (selected results in Table 3, see Table 6 for the full results). Inspecting the unprotected libraries shows that system libraries such as `libc` are never protected. Since these libraries are large and offer a variety of gadgets, they pose an attractive target for adversaries [113]. All protected binaries on the analyzed images depend on at least one of them.

Backwards-edge Protection on Android We find that there are two prevalent approaches for protecting return addresses in AArch64-based Android. First, there is the LLVM shadow call stack [6, 109]. Bionic allocates the shadow stack area during process creation, and it is only protected by information hiding through ASLR. Programs not using the shadow call stack will simply overwrite `X18` and ignore the allocated shadow stack area. As seen in Table 2, no binaries and barely any libraries are compiled with the shadow call stack. A probable explanation for its low usage is that a PA-based scheme is used instead, which does not share the weakness against memory disclosure attacks.

Comparably many binaries and libraries use the PA instructions `PACIBSP` and `AUTIBSP` to protect return addresses by tying them to the stack pointer (cf. Section 4.2.1). We also observe some cases with generic PA instructions such as `autia1716` (which authenticates the value in `X17` with `X16` as the context), but they appear only in stack unwinding code.

Development over Time The datasets for the S20 and the Mi 10 show that significant changes in CFI coverage occur only with new Android releases (cf. Table 2 and Table 5). Minor fluctuation in CFI coverage within the same Android version happens primarily due to the addition or removal of ELF files. However, some changes in the protection status of existing files, from protected to unprotected and vice versa, also happen in our dataset. Figure 4 shows the development over consecutive Android releases for the GSI. PA-based return address protection has been introduced with Android 12 and has since been extended, although the adoption rate slowed down with Android 14. The LLVM CFI coverage stagnates and, in some cases, even decreases with recent releases. Corresponding figures for the S20 and the Mi firmware can be found in Figure 6 in the appendix.

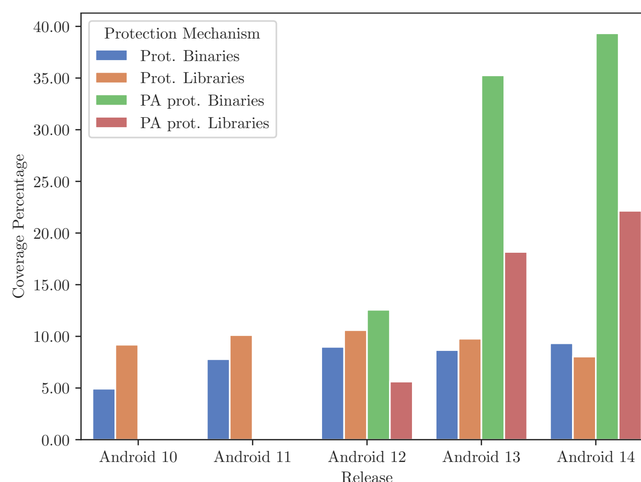


Figure 4: Development of mitigations on the GSI over time.

Memory-safe Code in Rust We find that Rust binaries are rather uncommon, with an average of only 14 files over all Android 13 firmware images. This might be subject to change as Google plans to primarily use Rust for new low-level code in Android [101]. Such a transition period comes with its own issues: Mixed binaries resulting from combined Rust and C / C++ codebases might be more vulnerable to memory corruption attacks because memory-safe parts can be abused to bypass mitigations such as CFI, which are deployed to protect unsafe code [74, 86]. Mixed binaries aside, Rust-based libraries will also contribute to the number of unprotected libraries and hence to the number of unprotected dependencies C / C++ binaries might have. While this issue could be addressed by compiling such libraries with LLVM CFI enabled, cross-language CFI support for Rust is not available yet [32].

Equivalence Class Size Frequencies For an indirect forward-edge control-flow transfer, the equivalence class size expresses the number of available call targets and hence the possible choices to an adversary. Therefore, the distribution of equivalence class sizes is a relevant metric to analyze type-based CFI schemes on a particular platform. Even though it shares the issue that the usefulness of targets is not con-

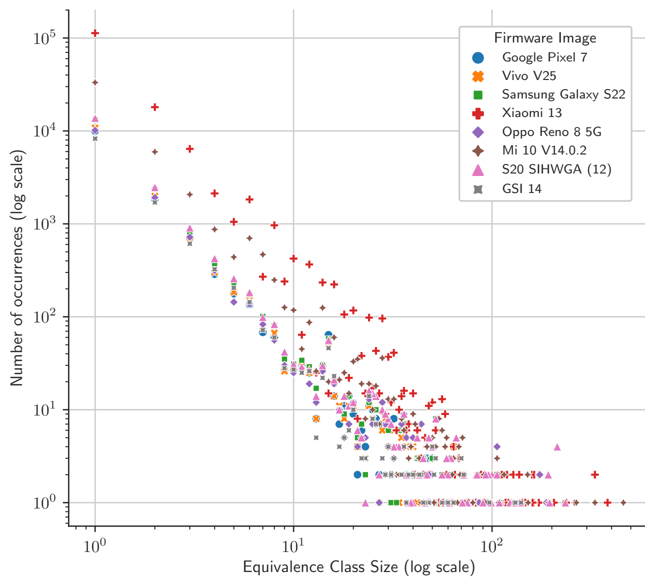


Figure 5: Equivalence class size distribution over all firmware.

sidered, it implicitly only considers full functions instead of arbitrary gadgets. Figure 5 depicts the frequencies with which each equivalence class size appears on the different firmware images. Equivalence classes are counted over all files on the corresponding firmware image as they were extracted from the `__cfi_check` function by our analysis. Equivalence classes consist of either jump tables, as used for checking function pointers, or vtables for checks involving C++ objects. Equivalence classes with the same type identifiers may appear multiple times if used in different files.

The distribution concentrates on single-member classes, with lesser frequencies of larger classes. This characteristic is desirable, as single-member classes imply that an adversary has no choice. We observe that the same outliers are often found across different firmware images due to the fact that they share a common codebase. Concerning the type of the tables these equivalence classes are based on, we found that jump tables are primarily responsible for the largest equivalence classes, with the exception of the Xiaomi 13 firmware.

Reachable Equivalence Classes Not all equivalence classes in a CFI-protected binary and its dependencies are actually reachable, i.e., there is no indirect control-flow transfer targeting them. Calls to `__cfi_slowpath` can be used to determine equivalence classes that are reachable from a binary. This approach is imperfect because `__cfi_slowpath` is also used for other CFI-related checks that are not strictly speaking indirect control-flow transfers. One instance of such a check is the `cfi-nvcall`, which checks non-virtual calls for polymorphic class types by checking the vtable pointer to ensure that the function is called with a compatible object [108]. This can be solved by following the CFG after the corresponding `__cfi_slowpath` call to determine the type of the next branch instruction, keeping only indirect calls. Such additional analysis steps introduce imprecision and runtime overhead, and we decided to focus on CFI checks independently of their purpose. To get an idea of how a program's dependencies increase the size of existing equivalence classes, we also included its dependencies for this particular analysis. Because this does not make sense for binaries with no protected dependencies, such files are ignored. Full results are depicted in Table 7 in the appendix. Besides for the Xiaomi firmware, the geometric mean of equivalence class sizes slightly decreases when considering dependencies. While this is contrary to the expectation that equivalence classes grow due to the merging of classes with common type identifiers, it can be explained by the fact that on these firmware images, protected libraries on average, have smaller equivalence classes than binaries. Therefore, the geometric mean decreases when also considering dependencies. The results restricted to reachable equivalence classes change as expected. Overall, reachable equivalence classes are expected to contain more than a single member because else there would be no need for function pointers or virtual functions. Table 7 contains equivalence class sizes below two because a significant portion (cf. Table 3) of dependencies are unprotected, and targets that are located in them cannot be considered.

5.2 Linux Study

The Linux kernel introduced support for Clang's CFI scheme in release 5.13 [111]. As a more efficient alternative, Linux also supports FineIBT, starting with version 6.2 [71]. FineIBT user-space support is also in the works, but it requires PLT format modifications, thus leading to ABI changes [44]. For backward-edge protection in user space, the Linux kernel 6.6 introduced CET shadow stack support on corresponding platforms [112]. Applications need to signal shadow stack compatibility by setting the `SHSTK` flag in the Executable and Linking Format (ELF) note, and the kernel must be configured with the `X86_USER_SHADOW_STACK` flag [60].

The build configuration of shipped kernels and applications usually depends on the Linux distribution. Because there are many different Linux distributions and their market share is hard to quantify [35], we exemplarily pick the most recent Debian, Ubuntu, Fedora Workstation, and Arch Linux releases as representatives of widespread distributions. As of September 2023, none of them ship a kernel recent enough to support the CET shadow stack, and none enables `CONFIG_CFI_CLANG` by default. On the application side, these distributions set the `-fcf-protection` build flag by default to produce shadow stack and IBT protected binaries, even though kernel support for neither feature is available yet [25, 26, 28, 53]. Clang's CFI scheme is not specified, probably because it is tied to Clang and unusable with other prevalent compilers such as GCC. On major Linux distributions, fully working mitigation combinations are not deployed yet. For this reason, we refrain from running a binary analysis study on Linux distributions.

Table 4: Windows 11 Insider Preview CFI coverage

File Type	Unprotected	Only WCFG	XFG	EC Size
Exe	2.68%	11.59%	85.73%	1.37 [G.M.]
DLL	2.62%	11.68%	85.70%	1.37 [G.M.]
Sys. DLL	0.91%	2.06%	97.04%	1.38 [G.M.]
Combined	2.63%	11.66 %	85.70%	1.37 [G.M.]

The Sys. DLL column covers all `.DLL` files located in `C:\Windows\System32\` and subdirectories thereof.

5.3 Windows Study

We study the Windows 11 Insider Preview developer build 23440 with respect to WCFG and XFG coverage. WCFG-related metadata in the Portable Executable (PE) header allows us to reliably detect WCFG and XFG usage. The former can be detected by checking the `DllCharacteristics` field [77], while XFG protection is indicated by the `GuardFlags` field of the load configuration. All XFG-instrumented functions are listed in the `GuardCFFunctionTable` in the load configuration and have the corresponding bit set in the flags part of their entry [75]. Hence, we use the following approach: First, we traverse the `GuardCFFunctionTable` and extract all entries with flag-bit 0x08 set. Then, for each entry, we extract the 8 bytes representing the type hash, which precedes the address indicated by its address part. As a result, we obtain the set of XFG-instrumented functions and their type hashes.

WCFG and XFG Coverage First, we measure WCFG and XFG coverage by enumerating all PE files compiled for x64 with either a `.dll` or `.exe` extension. We restrict files to these extensions to exclude files that share the PE format but are irrelevant to our study, such as `.mui` files used for multilingual user interfaces. Additionally, we ignore files without executable sections since they do not require protection. Virtual DLLs are one example of such files [124]. Table 4 gives an overview of the distribution of protection schemes. Contrary to the results from our Android study, the majority of analyzed files on Windows are compiled with XFG instrumentation.

Equivalence Class Sizes The geometric mean equivalence class size is similar to the one we observed on Android (cf. Table 7, second column), even though slightly lower. Besides Windows 11 being a codebase unrelated to Android, a contributing factor to this difference could be that not every protected function in an XFG-instrumented PE file is necessarily XFG protected, as WCFG can be used to protect individual functions. We found that, on average, 95.94% of `GuardCFFunctionTable` entries were marked as XFG protected for files with XFG instrumentation. This means that the on average remaining 4.06% of targets must be considered members of every XFG equivalence class in the corresponding file. The entire distribution of equivalence class sizes is depicted in Figure 7 in the appendix and looks similar to the distribution on Android (cf. Figure 5).

Backwards-edge Protection on Windows WCFG and XFG only protect forward-edge control-flow transfers. Microsoft tested a software-based shadow stack called Return Flow Guard but found it affected by information leakage attacks and an exploitable race condition [14]. Instead, they use hardware-supported schemes on supported platforms: On recent x86-based systems, the shadow stack of Intel's CET is used to protect backward-edge transfers [69] if the corresponding PE file sets the `CET_COMPAT` extended DLL characteristics bit.

On AArch64-based systems, recent Windows on ARM builds support PA for protecting return addresses [121]. Our analysis of the insider preview dev build 23419 yields a PA file coverage of 92 %. To calculate this, we enumerate all `.exe` and `.dll` files for AArch64 and search them for PA-related instructions. Then, we filter out cases of instructions that only appear incidentally in executable sections. The remaining instructions consist of the `PACIBSP` and `AUTIBSP` pair used for signing and authenticating the return address with the stack pointer as context and the `B` key, and the `XPACLRI` instruction for stripping PACs from the link register. Windows 11 on ARM uses the basic PA scheme, in which each return address is tied to the stack pointer value at function entry (refer to Section 4.2.1). Since better designs exist (e.g. [57,67]), it seems that the current implementation was deemed sufficiently secure, or the complexity or runtime overhead of such solutions was found unacceptable.

Bypassing XFG with Suppressed Functions WCFG supports function suppression, a feature to mark unsafe functions that should never be called indirectly [75]. Such functions are not placed into the WCFG function table and have no bitmap entry to mark them as valid functions. Developers can use this feature by using function modifiers in their code, and Microsoft uses it internally to protect system DLLs. In such DLLs, restricted functions are mostly related to control-flow tasks such as stack unwinding or exception handling.

We found that even though suppressed functions do not appear in the WCFG function table, they still have XFG type hashes. Consequently, they are valid call targets under XFG enforcement, as long as the corresponding call site has the same type hash. For suppressed functions of a sufficiently generic signature (i.e., no custom types appearing only in specific APIs), this implies that they can be reachable by an attacker, especially considering previously described techniques to reach such call sites [38,41]. We reported this issue to Microsoft and expect that Microsoft will fix this in future releases of Windows and their MSVC compiler by omitting XFG type hashes for suppressed functions.

6 Related Work

Various CFI schemes have been treated in previous research. Thereby lay the focus on either compatibility [123], a combination of precision, security, and performance [19], techniques applicable to resource-restrained embedded and real-

time processing devices [78], the precision of binary-level techniques [114], and the security boundaries of different approaches [66]. [81] introduces a framework for comparing different CFI policies. Since this framework operates on source code, it solves a different task than our analysis. Equivalent studies also exist for hardware-based schemes [33, 65, 105]. Because these surveys mostly compare academic prototypes, they do not address the usage of CFI in practice. In [116], CFI equivalence classes in the Linux kernel are analyzed. Their approach differs from ours, as they extract CFI targets by using an instruction pattern instead of symbolic execution. Consequently, they do not consider CDSO calls.

Several works exist that explore approaches to automated firmware analysis. `Firmalice` [98] detects authentication bypasses in binary programs automatically. It uses symbolic execution to detect such vulnerabilities based on a general and architecture-agnostic model characterizing them. Similarly, [22] employs full-system emulation of Linux-based firmware to identify vulnerabilities by checking accessible web pages, enumerating Simple Network Management Protocol (SNMP) information, and attempting known exploits.

Related to memory safety, the work in [125] runs a large-scale analysis to study the coverage of different mitigations in embedded-device firmware. To detect present mitigations, they use static indicators, including the occurrence of certain strings or symbols, the existence of specific ELF sections, or flags in the program header. However, they do not consider CFI usage. Targeting specifically the Android platform, [51] analyzes Android firmware to investigate its patch level. It builds on multiple static analysis tools to detect missing patches, app attribute misconfigurations, and cryptographic misuse. To perform static analysis tasks targeting pre-installed apps in Android firmware, the `FirmwareDroid` framework [103] was proposed. It has been applied to study advertiser tracker libraries shipped with pre-installed apps. [37] investigates the security of such pre-installed apps, focusing on privilege-escalation vulnerabilities by using a custom static taint analysis.

7 Recommendations for Improving CFI

A comparison of the state of the art CFI research with the schemes found in practice shows a large gap between scientific implementations and their adoption. Researchers identified compatibility as a long-standing issue [123]. When looking at the two instances where production systems and compilers have integrated results from research efforts [44, 110], it becomes evident that corresponding authors had direct ties to the industry. The corresponding financial backing and interest in creating solutions that are applicable to production systems could explain why these authors underwent the effort of submitting patches to LLVM and the Linux kernel.

The CFI schemes observed in practice are primarily coarse-grained. With LLVM's type-based scheme and the introduction of XFG, vendors are moving towards fine-grained schemes, which provide better security. This trend confirms that vendors are interested in moving forward and closing the gap between academia and industry.

Our analysis shows that equivalence classes of the fine-grained schemes tend to be small. Compared to coarse-grained schemes, this indicates a substantial improvement, but outliers exist and contribute to the choices of targets available to adversaries. Existing metrics for measuring CFI protection are insufficient to address this problem, therefore adding to the difficulty in evaluating the benefits of these mitigations.

We found that in the Android ecosystem, popular vendors rolled out CFI support differently over time. This indicates that even if there is a build environment that supports CFI and that is well-maintained and tested, adoption to real-world systems takes time. We strongly encourage vendors to ensure that CFI is applied to all binaries. From the vendor's perspective, system libraries should be shipped with CFI enabled to allow developers to benefit from compiling programs with CFI. Our tools will support vendors in analyzing their systems for potential gaps in CFI support, which might arise due to passing wrong compiler options in subprojects.

8 Conclusion

Our results show that CFI roll-out is not yet a finished process. We found the CFI coverage on Android lacking, especially regarding system-provided shared libraries. In these cases, CFI follows an all-or-nothing principle, meaning that security benefits are basically non-existent without a complete deployment. While the WCFG/XFG coverage we observed on Windows was better, it remains to be seen how long it takes until commercial off-the-shelf software builds are properly shipped with XFG protection once XFG is officially released.

With the increasing adoption of CFI, the hurdle for adversaries grows, who, in the best case, need to develop new exploitation techniques for each vulnerable program, hence raising the required effort and cost of attacks. In addition, specific bugs that would lead to arbitrary code execution without CFI can become unexploitable with CFI protection being applied, requiring adversaries to find stronger primitives. We hope to see CFI fully deployed in the future, along with more effective protection guarantees.

Acknowledgments

We thank the anonymous reviewers and the artifact evaluators for their helpful suggestions. This work has been funded by the German Research Foundation (DFG) in the project CRUST (grant number: 503199853).

Availability

The scripts described in the paper are published here: github.com/seemoo-lab/woot24_cfi_coverage_tools/

References

[1] Armv8.5-A memory tagging extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf. Whitepaper.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[3] National Security Agency. Ghidra. https://ghidra-sre.org/, March 2019.

[4] Android Open Source Project. AOSP - Control Flow Integrity. https://source.android.com/docs/security/test/cfi, 2022.

[5] Android Open Source Project. AOSP - Kernel Control Flow Integrity. https://source.android.com/docs/security/test/kcfi, 2022.

[6] Android Open Source Project. AOSP - ShadowCallStack. https://source.android.com/docs/security/test/shadow-call-stack, 2022.

[7] Android Open Source Project. AOSP - Homepage. https://source.android.com/, 2023.

[8] Android Open Source Project. AOSP - Arm Memory Tagging Extension. https://source.android.com/docs/security/test/memory-safety/arm-mte, 2024.

[9] AppBrain. Top android phone manufacturers. https://web.archive.org/web/20230317081510/https://www.appbrain.com/stats/top-manufacturers, 2023.

[10] Apple. Apple LLVM fork - pointer authentication documentation. https://github.com/apple/llvm-project/blob/d43163879bdb9576fff7a5a269d36920eee4ac29/clang/docs/PointerAuthentication.rst.

[11] Apple. Apple platform security. https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf, May 2022.

[12] ARM Holdings. Arm architecture reference manual for a-profile architecture. https://developer.arm.com/documentation/ddi0487/ha/?lang=en, February 2022.

[13] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[14] Joe Bialek. The evolution of CFI attacks and defenses. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2018_02_OffensiveCon/The%20Evolution%20of%20CFI%20Attacks%20and%20Defenses.pdf, 2018.

[15] Andrea Biondo, Mauro Conti, and Daniele Lain. Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets. In *NDSS*, San Diego, California, February 2018. Internet Society.

[16] Mark Brand. First handset with mte on the market. https://googleprojectzero.blogspot.com/2023/11/first-handset-with-mte-on-market.html, 2023.

[17] David Brash. Armv8-a Architecture: 2016 Additions. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-2016-additions, 2016.

[18] Michael D. Brown and Santosh Pande. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *Proceedings of the 12th USENIX Conference on Cyber Security Experimentation and Test*, CSET'19, page 5, USA, 2019. USENIX Association.

[19] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1), April 2017.

[20] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999, 2019.

[21] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow bending: On the effectiveness of Control-Flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association.

[22] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016.

[23] Long Cheng, Hans Liljestrand, Md Salman Ahmed, Thomas Nyman, Trent Jaeger, N. Asokan, and Danfeng Yao. Exploitation techniques and defenses for data-oriented attacks. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 114–128, 2019.

[24] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 38–49, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Arch Linux Community. makepkg.conf. https://gitlab.archlinux.org/archlinux/packaging/packages/pacman/-/blob/5fc0f6312b17abf707318c6909275721dab75a54/makepkg.conf.

[26] Debian Community. Debian dpkg-buildflags. https://manpages.debian.org/unstable/dpkg-dev/dpkg-buildflags.1.en.html.

[27] GrapheneOS Community. Grapheneos. https://grapheneos.org/.

[28] Ubuntu Community. Compilerflags - default flags. https://wiki.ubuntu.com/ToolChain/CompilerFlags.

[29] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, New York, NY, USA, 2015. Association for Computing Machinery.

[30] John Criswell, Nathan Dautenhahn, and Vikram Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*, pages 292–307, 2014.

[31] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian 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, San Diego, CA, August 2014. USENIX Association.

[32] Ramon de C Valle. Tracking issue for LLVM control flow integrity (CFI) support for rust. https://github.com/rust-lang/rust/issues/89653, 2023.

[33] Ruan de Clercq and Ingrid Verbauwhede. A survey of hardware-based control flow integrity (CFI). *CoRR*, abs/1706.07257, 2017.

[34] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of Path-Sensitive control security. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 131–148, Vancouver, BC, August 2017. USENIX Association.

[35] DistroWatch. Distrowatch page hit ranking. https://distrowatch.com/dwres.php?resource=popularity.

[36] Victor Duta, Fabian Freyer, Fabio Pagani, Marius Muench, and Cristiano Giuffrida. Let me unwind that for you: Exceptions to backward-edge protection. In *NDSS*, 2023.

[37] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. FIRMSCOPE: Automatic uncovering of Privilege-Escalation vulnerabilities in Pre-Installed apps in android firmware. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2379–2396. USENIX Association, August 2020.

[38] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 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 '15, pages 901–913, New York, NY, USA, 2015. Association for Computing Machinery.

[39] Francisco Falcon. How the MSVC compiler generates XFG function prototype hashes. https://web.archive.org/web/20230518085024/https://blog.quarkslab.com/how-the-msvc-compiler-generates-xfg-function-prototype-hashes.html, November 2020.

[40] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054, 2021.

[41] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. On the effectiveness of type-based control flow integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 28–39, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Andreas Follner, Alexandre Bartel, and Eric Bodden. Analyzing the gadgets. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639*, ESSoS 2016, page 155–172, Berlin, Heidelberg, 2016. Springer-Verlag.

[43] Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. CFInsight: A Comprehensive Metric for CFI Policies. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.

[44] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. Fineibt: Fine-grain control-flow enforcement with indirect branch tracking, 2023.

[45] Alex Gaynor. What science can tell us about C and C++'s security. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/, May 2020.

[46] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194, 2016.

[47] E.K. Goktas, A. Oikonomopoulos, Robert Gawlik, Benjamin Kollenda, I. Athanasopoulos, C. Giuffrida, G. Portokalidis, and H.J. Bos. Bypassing Clang's SafeStack for Fun and Profit. In *Black Hat Europe*, November 2016.

[48] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017.

[49] GRSecurity. Frequently asked questions about rap. https://grsecurity.net/rap_faq, 2023.

[50] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, 2014.

[51] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Chenglin Mao, Lingyun Ying, Song Liu, Xiaofeng Liu, Yuanzhi Li, Shanqing Guo, Meining Nie, and Haixin Duan. Can we trust the phone vendors? comprehensive security measurements on the android firmware ecosystem. *IEEE Transactions on Software Engineering*, 49(7):3901–3921, 2023.

[52] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1470–1486, New York, NY, USA, 2018. Association for Computing Machinery.

[53] Red Hat Inc. Using rpm build flags. https://src.fedoraproject.org/rpms/redhat-rpm-config/blob/f39/f/buildflags.md.

[54] Intel. Control flow enforcement technology. https://www.intel.com/content/dam/develop/external/us/en/documents/catc17-introduction-intel-cet-844137.pdf, December 2017.

[55] Intel. New intel vpro platform portfolio. https://www.intel.com/content/www/us/en/products/docs/processors/core/12th-gen-vpro-desktop-processors-brief.html, June 2022.

[56] Intel. Intel® 64 and ia-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2023.

[57] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Tightly seal your sensitive pointers with pactight, 2022.

[58] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1868–1882, New York, NY, USA, 2018. Association for Computing Machinery.

[59] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Safedispatch: Securing c++ virtual calls from memory corruption attacks. In *NDSS*, 01 2014.

[60] The kernel development community. Control-flow enforcement technology (cet) shadow stack. https://www.kernel.org/doc/html/v6.6-rc2/arch/x86/shstk.html.

[61] Mustakimur Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 195–211, Santa Clara, CA, August 2019. USENIX Association.

[62] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive call-site sensitive control flow integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 95–110, 2019.

[63] Hyungseok Kim, Junoh Lee, Soomin Kim, Seungil Jung, and Sang Kil Cha. How'd security benefit reverse engineers? : The implication of intel cet on function identification. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 559–566, 2022.

[64] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, October 2014. USENIX Association.

[65] Senyang Li, Weike Wang, Wenxin Li, and Dexue Zhang. Hardware-Based Software Control Flow Integrity: Review on the State-of-the-Art Implementation Technology. 11:133255–133280.

[66] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1835, New York, NY, USA, 2020. Association for Computing Machinery.

[67] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. Pacstack: an authenticated call stack, 2019.

[68] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, August 2019. USENIX Association.

[69] Jin Lin. Developer guidance for hardware-enforced stack protection. https://techcommunity.microsoft.com/t5/windows-kernel-internals-blog/developer-guidance-for-hardware-enforced-stack-protection/ba-p/2163340, 2021.

[70] Linus Torvalds. The linux kernel: extract-ikconfig. https://github.com/torvalds/linux/blob/6465e260f48790807eef06b583b38ca9789b6072/scripts/extract-ikconfig.

[71] LWN.net. Kernel release status. https://lwn.net/Articles/924113/.

[72] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 941–951, New York, NY, USA, 2015. Association for Computing Machinery.

[73] Connor McGarr. Exploit development: Between a rock and a (xtended flow) guard place: Examining XFG. https://web.archive.org/web/20231024102105/https://connormcgarr.github.io/examining-xfg/, August 2020.

[74] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-Language Attacks. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.

[75] Microsoft. Pe metadata. https://learn.microsoft.com/en-us/windows/win32/secbp/pe-metadata, June 2021.

[76] Microsoft. Control flow guard for platform security. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2022.

[77] Microsoft. Pe format. https://learn.microsoft.com/en-us/windows/win32/debug/pe-format, October 2022.

[78] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. Survey of Control-flow Integrity Techniques for Real-time Embedded Systems. 21(4):41:1–41:32.

[79] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. Drop the rop fine-grained control-flow integrity for the linux kernel. *Black Hat Asia*, 2017.

[80] Alan Mujumdar. Armv8.1-m pointer authentication and branch target identification extension. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension, 2021.

[81] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. Analyzing control flow integrity with LLVM-CFI. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, pages 584–597, New York, NY, USA, 2019. Association for Computing Machinery.

[82] Ben Niu and Gang Tan. Modular control-flow integrity. *SIGPLAN Not.*, 49(6):577–587, June 2014.

[83] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 914–926, New York, NY, USA, 2015. Association for Computing Machinery.

[84] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking holes in information hiding. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 121–138, Austin, TX, August 2016. USENIX Association.

[85] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851, 2021.

[86] Michalis Papaevripides and Elias Athanasopoulos. Exploiting mixed binaries. *ACM Trans. Priv. Secur.*, 24(2), January 2021.

[87] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-grained control-flow integrity through binary hardening. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164, Cham, 2015. Springer International Publishing.

[88] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224. USENIX Association, June 2003.

[89] Android Open Source Project. Increase the size of the shadow call stack guard region to 16mb. https://android-review.googlesource.com/c/platform/bionic/+/891622.

[90] Android Open Source Project. bionic. https://android.googlesource.com/platform/bionic, November 2022.

[91] Android Open Source Project. Configuring art. https://source.android.com/docs/core/runtime/configure, October 2022.

[92] LLVM Project. Branch target identification code-generation pass. https://github.com/llvm/llvm-project/commit/4bc81028d48c0ab07e7b429d2a98ed6d15140a23.

[93] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[94] Robert Schilling, Pascal Nasahl, and Stefan Mangard. Fipac: Thwarting fault- and software-induced control-flow attacks with arm pointer authentication. In Josep Balasch and Colin O'Flynn, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 100–124, Cham, 2022. Springer International Publishing.

[95] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten 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, 2015.

[96] Qualcomm Product Security. Pointer Authentication on ARMv8.3 - Design and Analysis of the New Software Security Instructions. Technical report, Qualcomm Technologies, Inc., 5775 Morehouse Drive, San Diego, CA 92121, U.S.A., January 2017. Available here: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[97] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[98] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, 2015.

[99] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[100] StatCounter. Operating system market share worldwide – jan 2020-jan 2023. https://gs.statcounter.com/os-market-share#monthly-202001-202301, March 2023.

[101] Jeffrey Vander Stoep. Memory safe languages in android 13. https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html, December 2022.

[102] Ke Sun, Ya Ou, Yahnhui Zhao, Xiaomin Song, and Xiaoning Li. Never let your guard down: Finding unguarded gates to bypass control flow guard with big data. https://www.youtube.com/watch?v=oD0rKvJcGbs, March 2017. BlackHat Asia 2017 conference talk.

[103] Thomas Sutter and Bernhard Tellenbach. FirmwareDroid: Towards Automated Static Analysis of Pre-Installed Android Apps. In *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 12–22, 2023.

[104] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, USA, 2013. IEEE Computer Society.

[105] Stefan Tauner and Mario Telesklav. Comparative analysis and enhancement of CFG-based hardware-assisted CFI schemes. *ACM Trans. Embed. Comput. Syst.*, 20(5s), September 2021.

[106] The Clang Team. Clang Documentation - SafeStack. https://clang.llvm.org/docs/SafeStack.html.

[107] The Clang Team. Clang documentation - control flow integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2023.

[108] The Clang Team. Control flow integrity design documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html, 2022.

[109] The Clang Team. Shadowcallstack. https://clang.llvm.org/docs/ShadowCallStack.html, 2023.

[110] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & LLVM. In *23rd USENIX security symposium (USENIX security 14)*, pages 941–955, 2014.

[111] Sami Tolvanen. Linux kernel - add support for clang cfi. https://github.com/torvalds/linux/commit/cf68fffb66d60d96209446bfc4a15291dc5a5d41.

[112] Linus Torvalds. Merge tag 'x86_shstk_for_6.6-rc1'. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=df57721f9a63e8a1fb9b9b2e70de4aa4c7e0cd2e.

[113] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, pages 121–141, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[114] Ruturaj K. Vaidya and Prasad A. Kulkarni. Assessing the effectiveness of binary-level cfi techniques. http://arxiv.org/abs/2401.07148, 2024.

[115] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 927–940, New York, NY, USA, 2015. Association for Computing Machinery.

[116] Jonathan Vexler. Characterization of forward-edge control-flow integrity targets in LLVM-compiled linux. https://cs.brown.edu/research/pubs/theses/masters/2020/vexler.jonathan.pdf, 2020.

[117] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC '15, pages 331–340, New York, NY, USA, 2015. Association for Computing Machinery.

[118] Wenhao Wang, Xiaoyang Xu, and Kevin W. Hamlen. Object flow integrity. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1909–1924, New York, NY, USA, 2017. Association for Computing Machinery.

[119] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. SafeHidden: An efficient and secure information hiding technique using re-randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1239–1256, Santa Clara, CA, August 2019. USENIX Association.

[120] David Weston. Advancing windows security. https://raw.githubusercontent.com/dwizzzle/Presentations/master/Bluehat%20Shanghai%20-%20Advancing%20Windows%20Security.pdf, 2019. Presented at Bluehat Shanghai 2019.

[121] David Weston. Mwc 2022: The next microsoft pluton device + pac technology. https://web.archive.org/web/20221104181427/https://blogs.windows.com/windowsexperience/2022/02/28/mwc-2022-the-next-microsoft-pluton-device-pac-technology/, February 2022.

[122] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. Warpattack: Bypassing CFI through compiler-introduced double-fetches, 2023.

[123] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. Confirm: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1805–1821, USA, August 2019. USENIX Association.

[124] Pavel Yosifovich, Mark E. Russinovich, Alex Ionescu, and David A. Solomon. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*. Microsoft Press, 7 edition, 2017.

[125] Ruotong Yu, Francesca Del Nin, Yuchen Zhang, Shan Huang, Pallavi Kaliyar, Sarah Zakto, Mauro Conti, Georgios Portokalidis, and Jun Xu. Building Embedded Systems Like It's 1996. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.

[126] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. ARM Pointer Authentication based Forward-Edge and Backward-Edge Control Flow Integrity for Kernels, 2019.

[127] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 337–352, USA, 2013. USENIX Association.

[128] Philipp Zieris and Julian Horsch. A leak-resilient dual stack scheme for backward-edge control-flow integrity. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 369–380, New York, NY, USA, 2018. Association for Computing Machinery.

[129] Changwei Zou, Yaoqing Gao, and Jingling Xue. Practical Software-Based Shadow Stacks on x86-64. *ACM Transactions on Architecture and Code Optimization*, 19(4):1–26, 2022.

[130] Changwei Zou, Xudong Wang, Yaoqing Gao, and Jingling Xue. Buddy stacks: Protecting return addresses with efficient thread-local storage and runtime re-randomization. *ACM Trans. Softw. Eng. Methodol.*, 31(2), mar 2022.

A Appendix

This appendix contains a discussion of how type-based CFI schemes can aid binary analysis tasks and a description and Proof of Concept (PoC) of the race condition in LLVM's CDSO CFI shadow mapping. Furthermore, it accommodates measurement data that did not fit into the main parts of the paper: Figure 6 shows how the CFI coverage changed over time with regards to the S20 and Mi 10 firmware images. Table 5 continues Table 2 with measurements for the Mi 10 firmware, Table 6 is an extended version of Table 3 covering all firmware images, Table 7 contains additional measurement data covering geometric means of equivalence class sizes in different categories, and Table 8 specifies the exact versions of the Android images we look at.

A.1 Using CFI to Aid Binary Analysis

Security aside, CFI can also unintentionally help analyze binary programs. For example, it has been shown that the `ENDBR` instructions used for Intel's IBT feature can be used to improve function boundary detection [63]. We argue that type-based schemes such as LLVM CFI or XFG also aid binary analysis by allowing to infer function addresses and information about their signatures. We propose the following approach:

1. Pre-compute type identifiers for common type signatures and classes and store them for fast look-up.
2. Find and annotate indirectly-callable functions with their corresponding type identifier. For LLVM CFI, the type identifier can be obtained by symbolically executing the `__cfi_check` function (details in Section 5.1.1). For XFG, this is done by extracting the type identifier that precedes the function.
3. Group annotated functions by their type identifiers.
4. Look up identifiers in the pre-computed data set, and if found, mark the function with the corresponding type. If some function in a set has a known signature, the same type can be applied to all other functions in the same set. The same principle can be applied to manually assigned signatures, which can also be propagated to functions in the same set.

A fundamental limitation of this approach is that only indirectly-callable functions can be analyzed, as only they will have the CFI-related metadata. More specifically, functions that are neither exported nor address-taken will not have jump tables or type hashes, respectively. A second issue is that type identifiers can collide, producing the same identifier for different types. Our approach would then produce wrong function signatures. Even though such collisions are unlikely, they are possible. However, we still think our approach is an interesting enhancement of typical binary analysis tools and leave a thorough evaluation for future work.

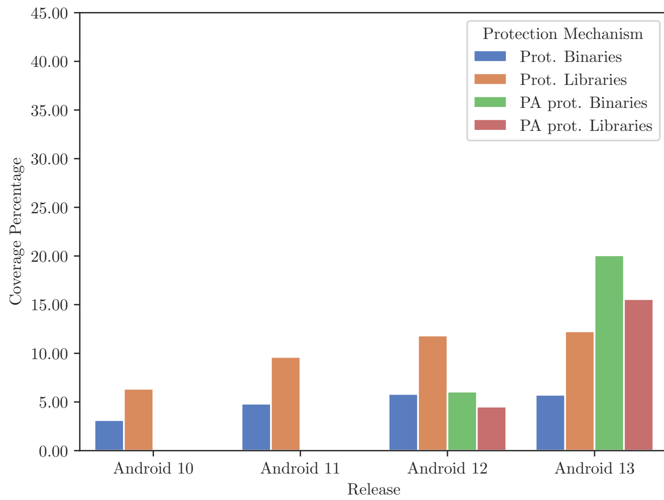
```
void thread_func(uintptr_t target) {
    // Simulate vulnerability to overwrite shadow mapping
         entry
    *reinterpret_cast<uintptr_t*>(target) =
        0xffffffffffffffff;
}

int main() {
    // Simulate leak of allocation address
    uintptr_t alloc = reinterpret_cast<uintptr_t>(
        mmap(nullptr,
             SIZE, PROT_READ | PROT_WRITE,
             MAP_PRIVATE | MAP_ANON | MAP_NORESERVE, 0,
                  0));
    std::cout << "Allocation at: "
              << std::hex << alloc << std::endl;

    // Calculate the address where the mapping entry for
    // target_func is located
    uintptr_t target = (alloc + ((reinterpret_cast<
         uintptr_t>(&target_func) >> kShadowGranularity)
         << 1) - DISTANCE);
    std::cout << "Target at: " << std::hex << target <<
         std::endl
              << "Shadow base at: "
              << std::hex << (alloc - DISTANCE) << std::
                   endl;

    // Start a thread to overwrite the target,
    // and trigger shadow mapping update
    std::thread t = std::thread(thread_func, target);
    // The .so file is arbitrary
    void *handle = dlopen("/usr/lib/p7zip/7z.so", 0);
    t.join();

    // Simulate an arbitrary write to redirect the
    // function pointer to the target_function
    int (*func_ptr)(int) = foo;
    func_ptr = reinterpret_cast<int (*)(int)>(&
         target_func);
    func_ptr(5);  // This call should fail under LLVM CFI
}
```

Listing 1: Proof of concept for CDSO CFI race condition bypass

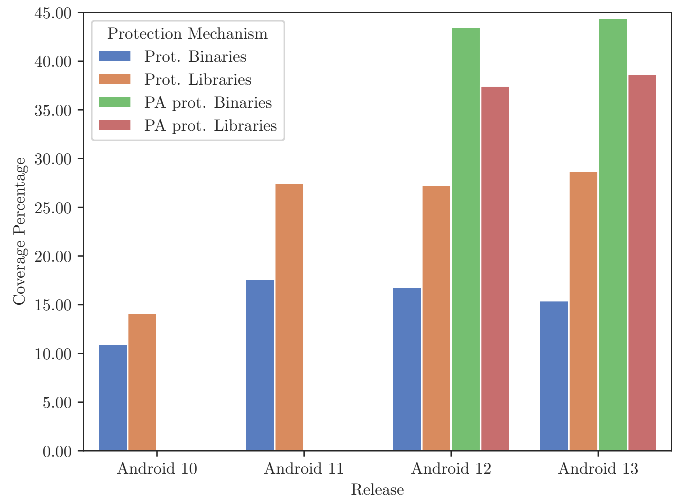
A.2 LLVM CDSO CFI race condition

LLVM's compiler runtime and Android's C standard library bionic [90] handle this by first allocating a new area for the shadow mapping, writing the desired values, and then re-mapping this new area to the previous shadow mapping address. This process opens up a timing window in which the newly allocated mapping is writable.

We successfully exploit this race condition to bypass CFI from within a C++ program, based on an experiment in which the adversary triggers a call to `dlopen` and then immediately starts a thread for writing to the shadow mapping afterward. A PoC for this bypass is shown in the appendix in Listing 1. Such issues indicate that an approach using embedded labels like Window's XFG is a more elegant solution, especially with respect to CDSO checks, but it is incompatible with execute-only memory.



(a) S20 Releases



(b) Mi 10 Releases

Figure 6: CFI coverage development over different Android releases. Bars represent the arithmetic mean over the analysed images within an Android release.

Table 5: CFI coverage of Android firmware images (continuation of Table 2)

	Files Total [count]			LLVM CFI Protected [%]				Shadow Stack Protected [%]			Pointer Auth. Protected [%]		
Vendor & Version	Binaries	Libraries	Kernel Modules	Binaries	Libraries	Kernel Modules	Kernel	Binaries	Libraries	Kernel Modules	Binaries	Libraries	Kernel Modules
Mi 10 2020-03-21 10	384	1987	41	10.94	14.09	0	✗	0	0.15	0	0	0	0
Mi 10 2020-07-15 10	382	1987	42	10.99	14.14	0	✗	0	0.15	0	0	0	0
Mi 10 2020-10-20 10	382	1992	42	10.99	14.11	0	✗	0	0.15	0	0	0	0
Mi 10 2021-01-10 11	383	1542	42	17.75	27.43	0	✗	0	0.19	0	0	0	0
Mi 10 2021-07-07 11	385	1534	42	17.66	27.57	0	✗	0	0.2	0	0	0	0
Mi 10 2022-01-20 11	385	1542	42	17.4	27.5	0	✗	0	0.19	0	0	0	0
Mi 10 2022-04-20 12	396	1681	40	16.67	27.31	0	✗	0	0.12	0	43.43	37.54	0
Mi 10 2023-01-12 12	397	1688	40	16.88	27.19	0	✗	0	0.12	0	43.58	37.38	0
Mi 10 2023-04-03 13	428	1668	40	15.42	28.72	0	✗	0	0.06	0	44.39	38.67	0
Mi 10 2023-05-17 13	428	1668	40	15.42	28.72	0	✗	0	0.06	0	44.39	38.67	0

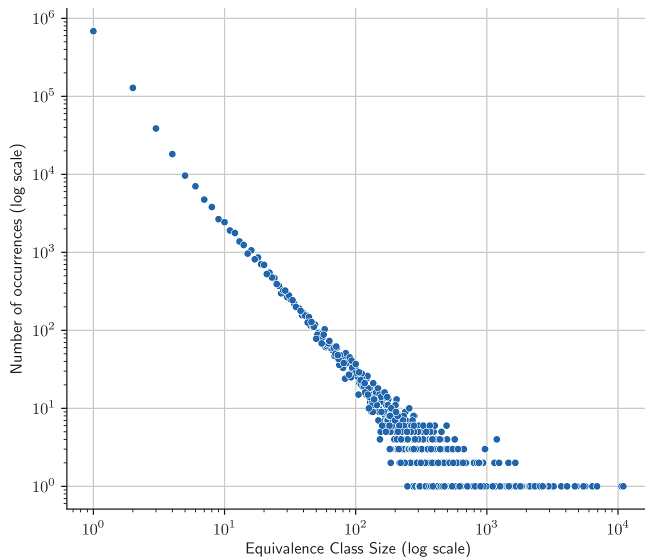


Figure 7: Equivalence class size distribution on the Windows 11 Insider Preview build 23440.

Table 6: Unprotected library dependencies in binaries.

Vendor & Device	Min [%]	Mean [%]	Max [%]
GSI 10	83.78	92.84	100.0
GSI 11	75.24	86.93	100.0
GSI 12	61.76	83.56	100.0
GSI 13	58.97	83.12	100.0
GSI 14	63.41	82.46	100.0
Xiaomi 13	24.14	75.83	100.0
Google Pixel 7	58.97	84.65	100.0
GrapheneOS Pixel 7	58.97	84.65	100.0
Oppo Reno 8 5G	61.76	87.0	100.0
Samsung Galaxy S22	60.98	87.13	100.0
Vivo V25	58.97	88.48	100.0
S20 2020-02-19	81.41	89.37	100.0
S20 2020-05-15	81.41	89.37	100.0
S20 2020-10-14	86.36	93.97	100.0
S20 2020-11-23	77.14	88.73	100.0
S20 2021-05-17	77.14	88.73	100.0
S20 2021-10-20	77.14	88.73	100.0
S20 2021-12-23	61.76	87.38	100.0
S20 2022-04-26	61.76	87.48	100.0
S20 2022-09-27	61.76	87.48	100.0
S20 2022-10-24	60.98	87.4	100.0
S20 2023-02-20	60.98	87.42	100.0
S20 2023-07-26	60.98	87.42	100.0
Mi 10 2020-03-21	77.97	93.81	100.0
Mi 10 2020-07-15	77.97	93.8	100.0
Mi 10 2020-10-20	77.97	93.8	100.0
Mi 10 2021-01-10	53.57	87.03	100.0
Mi 10 2021-07-07	53.57	87.0	100.0
Mi 10 2022-01-20	53.57	86.94	100.0
Mi 10 2022-04-20	49.36	87.14	100.0
Mi 10 2023-01-12	49.36	86.66	100.0
Mi 10 2023-04-03	47.49	86.13	100.0
Mi 10 2023-05-17	47.49	86.13	100.0

Table 7: Geometric mean of equivalence class sizes.

Firmware	All without deps.	All with deps.	Reachable without deps.	Reachable with deps.
GSI 10	1.52251	1.48887	1.61912	2.17204
GSI 11	1.53914	1.47219	1.74867	2.24668
GSI 12	1.5913	1.40786	1.81712	2.35598
GSI 13	1.5868	1.42263	1.91317	2.54419
GSI 14	1.55852	1.49571	1.88483	2.53512
Xiaomi 13	1.26808	1.31866	1.42438	2.06797
Google P7	1.47118	1.41269	1.74687	1.9628
Reno 8	1.47902	1.40975	1.73381	1.93443
Galaxy S22	1.44893	1.42762	1.7376	1.93009
Vivo V25	1.46451	1.42683	1.76094	1.99112
GrapheneOS	1.46739	1.41381	1.74092	1.95775
S20 2020-02-19	1.44735	1.43	1.60504	1.76194
S20 2020-05-15	1.44887	1.43365	1.60284	1.76729
S20 2020-10-14	1.44793	1.47095	1.60507	1.71205
S20 2020-11-23	1.46341	1.46212	1.64564	1.81048
S20 2021-05-17	1.46006	1.4641	1.64128	1.81521
S20 2021-10-20	1.46704	1.46407	1.64932	1.82339
S20 2021-12-23	1.47126	1.4107	1.67733	1.8489
S20 2022-04-26	1.47519	1.41253	1.68565	1.85759
S20 2022-09-27	1.47654	1.41281	1.68332	1.84906
S20 2022-10-24	1.48012	1.42268	1.71563	1.89752
S20 2023-02-20	1.47762	1.42254	1.70653	1.8927
S20 2023-07-26	1.47624	1.42209	1.71574	1.89804
Mi 10 2020-03-21	1.34242	1.3812	1.38852	1.55355
Mi 10 2020-07-15	1.3431	1.37911	1.38709	1.54917
Mi 10 2020-10-20	1.33849	1.37826	1.38092	1.54222
Mi 10 2021-01-10	1.38289	1.41024	1.45177	1.65537
Mi 10 2021-07-07	1.3849	1.41137	1.45473	1.65985
Mi 10 2022-01-20	1.36369	1.40777	1.42972	1.6782
Mi 10 2022-04-20	1.33301	1.37841	1.40426	1.53657
Mi 10 2023-01-12	1.33273	1.3783	1.39999	1.53187
Mi 10 2023-04-03	1.34791	1.37218	1.44178	1.66612
Mi 10 2023-05-17	1.34805	1.37145	1.4436	1.67142

Table 8: Analysed Android Firmware Images

Vendor	Device	Android	Release Date	Firmware Identifier	URL (full URL on-hover)
GSI	n/a	10	October 2019	gsi_gms_arm64-exp-QJR1	dl.google.com
GSI	n/a	11	September 2020	gsi_gms_arm64-exp-RP1A	dl.google.com
GSI	n/a	12	July 2022	gsi_gms_arm64-exp-SQ3A	dl.google.com
GSI	n/a	13	April 2023	gsi_gms_arm64-exp-T3B3	dl.google.com
GSI	n/a	14	August 2023	gsi_gms_arm64-exp-UPB5	dl.google.com
Google	Pixel 7	13	March 2023	panther-t3b2.230316.003-factory-c65097bc	dl.google.com
GrapheneOS	Pixel 7	13	September 2023	2023091800	releases.grapheneos.org
Vivo	V25	13	January 2023	PD2215F_EX_A_13.1.13.5.W30.V000L1	in-sysup-txdl.vivoglobal.com
Samsung	Galaxy S22	13	January 2023	S901BXXU3CWAI_S901BOXM3CWAI_EUX	www.sammobile.com
Xiaomi	Xiaomi 13	13	February 2023	fuxi_eea_global_V14.0.15.0.TMCEUXM	bigota.d.miui.com
Oppo	Reno 8 5G	13	November 2022	CPH2359_MT6893_EX_11_A.18_221121	oppostockrom.com
Samsung	Galaxy S20	10	February 2020	G980FXXU1ATBM	samfw.com
Samsung	Galaxy S20	10	May 2020	G980FXXU2ATE6	samfw.com
Samsung	Galaxy S20	10	October 2020	G980FXXU5BTJ3	samfw.com
Samsung	Galaxy S20	11	November 2020	G980FXXU5CTKG	samfw.com
Samsung	Galaxy S20	11	May 2021	G980FXXS8DUE4	samfw.com
Samsung	Galaxy S20	11	October 2021	G980FXXSCDUJ5	samfw.com
Samsung	Galaxy S20	12	December 2021	G980FXXSCEUL7	samfw.com
Samsung	Galaxy S20	12	April 2022	G980FXXUEFVDB	samfw.com
Samsung	Galaxy S20	12	September 2022	G980FXXSFFVIB	samfw.com
Samsung	Galaxy S20	13	October 2022	G980FXXUFGVJE	samfw.com
Samsung	Galaxy S20	13	February 2023	G980FXXSFHWB1	samfw.com
Samsung	Galaxy S20	13	July 2023	G980FXXSIHWGA	samfw.com
Xiaomi	Mi 10	10	March 2020	V11.0.9.0.QJBEUXM	bigota.d.miui.com
Xiaomi	Mi 10	10	July 2020	V11.0.18.0.QJBEUXM	bigota.d.miui.com
Xiaomi	Mi 10	10	October 2020	V12.0.6.0.QJBEUXM	bigota.d.miui.com
Xiaomi	Mi 10	11	January 2021	V12.2.4.0.RJBEUXM	bigota.d.miui.com
Xiaomi	Mi 10	11	July 2021	V12.5.2.0.RJBEUXM	bigota.d.miui.com
Xiaomi	Mi 10	11	January 2022	V12.5.8.0.RJBEUXM	bigota.d.miui.com
Xiaomi	Mi 10	12	April 2022	V13.0.4.0.SJBEUXM	bigota.d.miui.com
Xiaomi	Mi 10	12	January 2023	V13.0.10.0.SJBEUXM	bigota.d.miui.com
Xiaomi	Mi 10	13	April 2023	V14.0.1.0.TJBEUXM	bigota.d.miui.com
Xiaomi	Mi 10	13	May 2023	V14.0.2.0.TJBEUXM	bigota.d.miui.com