Twenty years later: Evaluating the Adoption of Control Flow Integrity

SABINE HOUY and ALEXANDRE BARTEL, Umeå University, Sweden

Memory corruption vulnerabilities still allow compromising computers through software written in a memory-unsafe language such as C/C++. This highlights that mitigation techniques to prevent such exploitations are not all widely deployed. In this paper, we introduce SeeCFI, a tool to detect the presence of a memory corruption mitigation technique called control flow integrity (CFI). We leverage SeeCFI to investigate to what extent the mitigation has been deployed in complex software systems such as Android and specific Linux distributions (Ubuntu and Debian). Our results indicate that the overall adoption of CFI (forward- and backward-edge) is increasing across Android versions (~30% in Android 13) but remains the same low (<1%) throughout different Linux versions. Our tool, SeeCFI, offers the possibility to identify which binaries in a system were compiled using the CFI option. This can be deployed by external security researchers to efficiently decide which binaries to prioritize when fixing vulnerabilities and how to fix them. Therefore, SeeCFI can help to make software systems more secure.

CCS Concepts: • **Security and privacy → Operating systems security**.

Additional Key Words and Phrases: static analysis, memory corruption vulnerabilities, mitigation techniques, CFI, software maintenance

1 INTRODUCTION

Memory corruption vulnerabilities are still among the most critical software bugs. They exist since memory-unsafe programming languages such as C/C++ are still commonly used to write code for highly used software such as web browsers or device drivers. These vulnerabilities include buffer and integer [24] overflows – and, more generally, all out-of-bound accesses –, uninitialized memory, type confusion, and use-after-free. We illustrate how attackers could leverage such vulnerabilities to execute arbitrary code through two cases. The first is CVE-2023-2731 [12, 40], a vulnerability in the `libxml2` library, a widely used XML parsing library in various software applications across many operating systems, including Ubuntu and Debian. The issue lies in the handling of XML documents during the parsing process. When the library attempts to free or deallocate memory that has been incorrectly managed, it may overwrite adjacent memory areas on the heap. This can lead to arbitrary code execution if an attacker can craft an XML document that exploits this overflow. This vulnerability is particularly dangerous because it allows attackers to potentially execute malicious code with the privileges of the application using `libxml2`. The second is CVE-2022-20127 [39], an out-of-bounds write due to a double free. This could lead to remote code execution with no additional execution privileges needed and no user interaction required for exploitation. It affects Android-10, Android-11, Android-12, and Android-12L and has a score of 10, the highest possible score for a CVE. These two examples illustrate that while techniques have been developed to prevent the exploitation of specific kinds of vulnerabilities, including out-of-bound writes [62], it seems that not all of them have been optimized to be widely deployed.

One of the first memory corruption attacks described in the literature consists of smashing the stack to change the control flow of the program [49]. The stack has been made non-executable to prevent stack smashing attacks, and this technique is widely deployed [3]. However, a new attack technique, Return-Oriented Programming (ROP) for leveraging gadgets, leveraging gadgets, has been introduced to bypass this mitigation technique. The code is randomized in memory

Authors' address: Sabine Houy, sabine.houy@umu.se; Alexandre Bartel, alexandre.bartel@cs.umu.se, Umeå University, Universitetstorget 4, Umeå, Sweden, 90187.

using another mitigation technique called ASLR [20, 50, 81] to prevent ROP attacks. It is a cat-and-mouse game: new mitigation techniques must be implemented when new attack vectors are found. However, not all mitigations are deployed. One potential cause could be the high-performance impact, resulting in a negative user experience. Another cause could be that integration requires human effort to update the software configuration to compile correctly, increasing the testing time and, thus, the deployment time. More recent mitigation techniques, such as Control Flow Integrity (CFI), aim at detecting and blocking the exploitation of vulnerabilities at runtime [2, 76]. The idea is to inject additional code in binaries at compilation time. One of the aims of this additional code is to check that branch targets are consistent. For instance, at the assembly level, a branching instruction `call rax`, in the context of a virtual call on an Object A, can only call a method defined in class A or a subclass. If an attacker has tampered with the value of register `rax`, the additional code will detect and stop the program to prevent exploitation. This is also called forward-edge CFI, while the protection of function returns is called backward-edge CFI. In practice, backward-edge CFI is named ShadowCallStack.

Some mitigation techniques are easy to detect. For instance, a PIE (position-independent executable) binary, similar to the ASLR mitigation technique, will have a specific flag in its header. A non-executable stack is detected by checking the configuration at the operating system level. However, there is no straightforward way to detect the presence of CFI in a binary. Indeed, CFI is compiler-dependent and injects additional code within the binary at compilation time. Unfortunately, to the best of our knowledge, the compiler does not add any flag in the header of the compiled binaries with CFI, leaving the detection of CFI to program analysis. In this paper, we present SeeCFI, which detects the presence of CFI within a binary. In particular, we focus on detecting CFI injections for forward-edge virtual call protection and the general detection of backward-edge CFI. SeeCFI is the implementation of a generic approach detecting CFI. Nevertheless, specific information on how a compiler injects CFI must be known well to accurately detect CFI code chunks in a binary. Currently, SeeCFI can statically analyze binaries compiled with Clang and GCC, two of the main C/C++ compilers existing today. We leverage SeeCFI to analyze native Android, Samsung, GrapheneOS (an Android fork focusing on security and privacy), and the Linux distributions Ubuntu and Debian to understand to what extent real-world state-of-the-art systems implement and deploy CFI. Our contributions are the following:

(1) *Implementation of SeeCFI* to detect whether a binary was compiled using forward- or backward-edge CFI. Our evaluation shows that SeeCFI has a very low false positive (<0.004) and false negative rate (<0.17). SeeCFI is open-source and can be found at *https://github.com/software-engineering-and-security/SeeCFI*.
(2) *Evaluation of the adoption of forward- and backward-edge CFI enforcement* throughout different Android and Linux versions. Our results show that in Android, the deployment of CFI is increasing even though it is, at this point, still low (~28.7%). On the other hand, the deployment of CFI in Linux systems is almost nonexistent (<1%) and not increasing, leaving the system vulnerable to control-flow hijacking attacks.

The remaining part of the paper is organized as follows. Section 2 presents the necessary information regarding Control Flow Integrity. Then, in Section 3, we describe the approach to detecting CFI, along with some technical details regarding SeeCFI's implementation. In Section 4, we give an overview of our data collection and experimental setup. We present our results in Section 5 and evaluate the performance of SeeCFI, including a discussion giving explanations that connect to the limitations of CFI described in Section 6. Related work is covered in Section 7. Finally, we conclude and present future work in Section 8.

2 BACKGROUND

Control Flow Integrity (CFI) was first introduced in 2005 in the initial paper by Abadi et al., which was revised in 2009 [2]. However, there was no common and widely adoptable implementation for almost one decade. Most custom implementations had too much overhead to be applicable [27]. In 2014, Tice et al. published the first integration into the production compilers GCC and LLVM [76]. CFI was introduced to protect low-level languages like C and C++ against memory corruption vulnerabilities, allowing control-flow hijacking attacks. Attackers can exploit these vulnerabilities (e.g., a buffer overflow or integer overflows [32]) to change the behavior of a program and even to gain full control over the control flow of a program [68].

Buffer overflows occur when more data is written into a buffer than its actual length. In general, if a non-maliciously triggered buffer overflow occurs, the program will either crash, wrongly execute, or execute normally. However, attackers can exploit *buffer overflow vulnerabilities* to perform stack smashing and heap smashing attacks depending on where the overwritten buffer is stored in memory. Stack-based buffer overflow vulnerabilities are prominent targets for attacks. This enables an attacker to compromise code, data, or the return address pointer, as is most common in attacks. An attacker has different options to exploit this vulnerability, but most focus on overwriting the return address pointer saved on the stack. If the stack is executable, an attacker can overwrite the buffer with their own malicious code and set the return address pointer to the beginning of the buffer. Another option is to overwrite the pointer with the address pointing to the malicious code, which can be achieved by a *code-reuse attack*[1] or a *ret2libc attack*[2]. When the return address pointer is overwritten with another address, the control flow will be redirected to this address instead of returning as intended by design. Several techniques exist to prevent specific *control-flow hijacking* attacks, and one of the most promising ones is Control Flow Integrity.

2.1 Control Flow Integrity

Control Flow Integrity (CFI) is intended to support or extend other security mechanisms, e.g., ASLR, by verifying the targets of indirect function calls, jumps, and returns. There are two types of CFI: (i) forward-edge enforcement concerning indirect function calls and jumps (Section 2.1.1), (ii) and backward-edge dealing with the returns of previously performed function calls (Section 2.1.2). Even though there are different implementations, the basic structure is always similar.

2.1.1 Forward-edge enforcement. It consists of two components. The first is to create a control flow graph (CFG) of the program in question to identify the relevant parts of the code, done by a static analysis approach. The compiler identifies indirect function calls, jumps, and all their possible destinations. On the call site, it injects code responsible for verifying the branching instruction. This means the pointer pointing to the call or jump target must be in the set of valid destinations. If this check fails, the program crashes. As the second component of forward-edge CFI, the check itself occurs at runtime through a dynamic enforcement mechanism. By that, the program's control flow is restricted according to the generated CFG.

Window's compiler Visual Studio [37], GNU's compiler GCC [31], and LLVM's compiler Clang [71] implement this process slightly differently. GCC's and Clang's implementations are both based on Tice et al. [76]. Windows added the extension Control Flow Guard (CFG) in 2015 [37]. Android has used one of Clang's forward-edge implementations to protect its kernel code and parts of its user-space components since 2018.

[1] *Code-reuse attacks* involve exploiting vulnerabilities to reuse existing code in memory to perform malicious actions, avoiding detection by security mechanisms.

[2] *ret2libc* is a type of code-reuse attack where an attacker redirects a program's control flow to functions in the C standard library (`libc`), often bypassing protections like non-executable memory.

```c
1  struct auth {
2   char p[AUTHMAX]; // pwd
3   void (*func)(struct auth*);
4  };
5
6  void success() {
7   printf("Auth. ok.\n");
8  }
9
10 void failure() {
11  printf("Auth. failed\n");
12 }
13
14 void auth(struct auth *a){
15  if (strcmp(a->p, "p") == 0)
16   a->func = &success;
17  else
18   a->func = &failure;
19 }
20
21 int main(int ac, char **av){
22  struct auth a;
23  a.func = &auth;
24  printf("Your pwd:\n");
25  scanf("%s", &a.pass);
26  a.func(&a);
27 }
```

(a)

```
1  <main>:
2  ...
3
4  call QWORD PTR [rbp-0x18] (= &auth)
5
6  ...
7
```

(b)

```
1  <main>:
2  ...
3
4  mov rax, QWORD PTR [rbp-0x8] (= &auth)
5  movabs rcx, &typeid_auth
6  cmp rax, rcx
7  je check_passed
8  ud2
9  ...
10
11 check_passed:
12 call rax
13 ...
14
15 <typeid_auth>:
16 jmp &auth
17 ...
```

(c)

Fig. 1. C-Program source code (a), assembly instructions without CFI (b), and assembly instructions with forward-edge CFI (c)

Figure 1 illustrates the difference in assembly code when compiled with and without forward-edge CFI; (a) displays the C source code of the compiled file, (b) illustrates the relevant parts of the assembly code compiled without the CFI option, and (c) shows the corresponding assembly code parts when compiled with forward-edge CFI. In line 26 (a) is the function's call that needs to be protected by CFI. It needs protection as this is an indirect call, which is based on an address and not a concrete value. Therefore, an attacker could tamper with this value, changing it to a malicious destination. Line 4 in (b) shows this call in assembly without CFI. The address to function `auth` is stored in `rbp-0x18`. In illustration (c), we can see how the assembly code changes when compiled with CFI. In line 4, the address of function `auth` is loaded into register `rax`, and then the address to the `typeid` of the function `auth` is moved into register `rcx`. The `typeid` of a function is its signature, which is generated based on its return type and parameter types upon compilation. Based on this signature, the compiler generates a set of valid call targets. In line 6, registers `rax` and `rcx` are compared, meaning that the value of the register to be called (`rax`) is compared with the validation value, the `typeid` (`rcx`). If the check passes, the program jumps from line 7 to line 12 to call the function `auth`. Otherwise, the program crashes by executing `ud2` (line 8).

2.1.2 Backward-edge enforcement. Backward-edge protection aims to protect the return address after returning from a previous function call. Moreover, it is less researched (fewer publications)

than forward-edge. Windows introduced hardware-enforced stack protection [35] based on Intel's Control-flow Enforcement Technology (CET) in 2019 as a compiler plugin. GCC's [31] and Clang's [73] implementation is called *ShadowCallStack* and is currently only supported for aarch64. When using *ShadowCallStacks*, a shadow stack is created in addition to the "regular" program stack. The return address is then pushed on both stacks, and both values are compared when popped in the function's epilogue. If the values do not match, the program execution terminates or crashes [11].

LLVM removed the implementation of backward-edge CFI (ShadowCallStack) from version 9.0 onwards for x86_64 due to race-conditions enabled by return semantics [74]. These do not affect aarch64 (arm64); thus, backward-edge CFI is currently only available for aarch64. The call-return semantics of x86_64 must be changed to prevent the race condition [74]. However, that would introduce an unacceptable performance overhead due to return branch predictions [74, 75]. Since Android is mainly deployed on aarch64-based devices, the runtime implementation of *ShadowCallStack* was added to its `libc` (`bionic`) [54] in 2019.

3 DETECTING CFI IN BINARIES

This section describes our approach to identifying forward- and backward-edge CFI, their implementations, and our test setup and data collection.

3.1 Approach

We used a manual approach to understand how LLVM's CFI implementation works in practice. In our approach, we only focus on indirect and virtual calls and assume that each binary contains at least one. We compiled C and C++ files both with and without the CFI flag and then compared the disassembled results.

Based on these findings, we implement the SeeCFI tool, which serves the purpose of identifying whether a binary was compiled using CFI. Furthermore, SeeCFI returns whether the *multi-module* CFI (cross-DSO), see Section 3.1.2, or *single-module* CFI option, described in Section 3.1.1, was used. In addition to detecting the presence of forward-edge CFI, SeeCFI can also determine whether the binary deploys backward-edge CFI in the form of a ShadowCallStack. The tool is composed of two main parts. The first one contains the forward-edge CFI detection, including the identification of the option used, and the second one is the detection of backward-edge CFI.

We perform the analysis on the binary level, as not all images are open-source; thus, only the binaries are available. For instance, Samsung images and Pixel[3] images are closed-source. Therefore, we use a Makefiles- and Blueprints-based analysis approach to evaluate the performance of SeeCFI on AOSP only. Section 4.3 explains this approach in more detail.

3.1.1 Single-module CFI. Single-module CFI refers to the initial purpose of forward-edge CFI: to protect virtual and indirect calls. This is implemented by adding checks before performing such a call. The checks verify that the target of the call is in a set of predefined valid destination addresses. The compiler defines this set by generating a CFG and identifying all functions with the same signature as the call target. The signature is based on the return and argument types of a function. The check is implemented as the function `llvm.type.test`. This function checks that the function to be called has the same *type* as the valid call targets, meaning the same return type and the same parameter types. The check itself is performed during runtime; if the check fails, the execution aborts using an `ud` or `brk` instruction. The check can directly compare the expected call destination and the call target or indirectly with some calculations beforehand based on both values.

[3]The Pixel images are based on AOSP, which is open-source, but they also contain additional close-sourced binaries, e.g., related to the PlayStore.

3.1.2 Multi-module CFI (cross-DSO). Multi-module CFI is an extension to the single-module CFI implementation to support shared libraries. Dynamic Shared Objects (DSO) are object files used simultaneously by multiple programs, reducing the necessity of duplicated code and thus decreasing memory usage. It implements all policies of the single-module CFI with the addition that it applies these across DSO boundaries. Multi-module CFI consists of two functions. The first is implemented in each module (DSO), and the second is in a runtime support library. The function `__cfi_check()` is injected into each DSO at compilation time. This function offers external modules CFI checks to targets inside itself, as no outside function can determine which function calls are valid within another DSO. The second function is called `__cfi_slowpath()` and resides within a shared library such as `libdl.so` in Android. It is responsible for finding the correct `__cfi_check()` function to perform the CFI check triggered from another DSO.

3.1.3 Backward-edge CFI (ShadowCallStack). ShadowCallStack is LLVM's implementation for the aarch64 architecture, as explained in Section 2. It protects the return address of a function call from overwrites by adding a check in the function's epilogue before returning. For this purpose, the return address is additionally stored on the ShadowCallStack, implemented in the function's prologue (`str x18, <`*ret_addr*`>`). Before returning, the return address and the address stored on the ShadowCallStack (`ldr <`*ret_addr*`>, x18`) are compared. If they match, they pass the check, and the function can return. Otherwise, the program crashes. Register `x18` is treated as a special case register and used as the *shadow-call-stack-register*.

3.2 Forward- & Backward-edge CFI Detection

This section explains the algorithms used in SeeCFI. In Section 3.2.1, we start by explaining the detection of forward-edge CFI split into two algorithms. The first one detects multi-module CFI (Algorithm 1), and the second one checks for single-module CFI (Algorithm 2). SeeCFI only checks for single-module CFI if it cannot detect multi-module CFI. Then, in Section 3.2.2, we explain how we detect the use of backward-edge CFI.

3.2.1 Forward-edge CFI Detection. Firstly, Algorithm 1 validates whether multi-module CFI is used, as this only requires checking the presence of the two functions mentioned before (`__cfi_slowpath` and `__cfi_check`). The function `__cfi_check` must be present within the analyzed binary. It first loads the binary to analyze using a binary analyzer framework. Then, the framework's symbol `loader` determines which of the CFI symbols (line 2) is present. Either the `__cfi_check` is part of the analyzed binary, or both symbols are present to be verified as CFI enabled. If both symbols exist (line 3), then the binary is compiled using the "cross-DSO" extension (line 4), and the algorithm returns. If the symbols do not exist in the binary, Algorithm 2 is called (line 7).

Algorithm 1 Check the presence of multi-module CFI (cross-DSO)

```
1: procedure CHECK_MULTI_MODULE_CFI(binary)
2:     symbols ← proj.LOADER.FIND_SYMBOL(cfi_symbols)
3:     if symbols then
4:         multi_module_cfi ← True
5:     else
6:         multi_module_cfi ← False
7:         CHECK_SINGLE_MODULE_CFI
```

Algorithm 2 then generates the CFG for the whole binary and extracts all relevant branching nodes (line 2). A branching node is considered relevant if it has two successors, indicating a

conditional jump. It then iterates over all the relevant nodes, checking if the successors contain the required instructions (line 4). One successor has to contain a system interrupt instruction, such as `ud` for ARM, and the other a `call` instruction. If both instructions are present in the two branches, the function `check_cmp()` is called (line 5) to verify if the comparison is based on the register used in the `call` instruction. The comparison is directly based on the register used in `call` (line 9). In that case, the algorithm terminates with the result that single-module CFI is enabled (line 10). It could also be indirectly based on the call register, which requires further checks (line 12). The registers used in the `cmp` statement are traced back to identify whether one of them was assigned to the content of the register used in the `call` (line 15). If so, the algorithm concludes that the binary was compiled using single-module CFI (line 16). Otherwise, the functions return to the fallthrough that single-module CFI was not used (line 6).

Algorithm 2 Check the presence of single-module CFI

```
1:  procedure CHECK_SINGLE_MODULE_CFI
2:      branching_nodes ← cfg.GET_BRANCHING_NODES
3:      for branching_nodes do
4:          if cfi_check_failed && call_instruction in successor then
5:              CHECK_CMP(n, s2)
6:      single_module_cfi ← False
7:
8:  procedure CHECK_CMP(n, s2)
9:      if check_statement exists && based on call_reg then
10:         single_module_cfi ← True
11:     else
12:         TRACK_REG(code, call_reg)
13:
14: procedure TRACK_REG(code, call_reg)
15:     if reg in cmp_regs assigned to call_reg then
16:         single_module_cfi ← True
17:         single_module_cfi ← False
```

There are three possible output options when detecting that a binary is compiled using forward-edge CFI. The first belongs to multi-module CFI (cross-DSO) identification, and the other to single-module CFI. When looking for single-module CFI, two possibilities exist: Either the register containing the address to the protected call target is directly used or indirectly, meaning that the comparison is based on the value of the target (its address) but using another register with possible calculations before.

3.2.2 Backward-edge CFI Detection. Algorithm 3 verifies the presence of backward-edge CFI implemented as the ShadowCallStack. As this detection is also based on code patterns, such as the detection of the single-module CFI, the algorithm needs the generated CFG. From the CFG, it obtains all functions present in the binary and iterates over them (line 2). Only functions that return are of relevance to us. For each relevant function, Algorithm 3 determines the entry block (first basic block of the function) and all return blocks. One function can have more than one basic block containing the `ret` instruction. The entry block is then checked to contain the `str` instruction in combination with register `x18` (line 4). The entry block corresponds to the function's prologue and is responsible for storing the return address on the ShadowCallStack. If the prologue passes this

check (line 6), the return blocks are checked if they contain the `ldr` instruction combined with the *shadow-call-stack-register* (`x18`) (line 7). The return blocks correspond to the function's epilogue. If this condition is also true, the algorithm returns "`shadow_call_stack = True`" (line 8), otherwise "`shadow_call_stack = False`"(line 9).

Algorithm 3 Check the presence of backward-edge CFI (ShadowCallStack (SCS))

```
1: procedure CHECK_SHADOWCALLSTACK
2:     for addr, func in all_functions do
3:         prologue ← False
4:         if scs_reg is stored in entry_block then          ▷ scs_reg = register to store SCS
5:             prologue ← True
6:         if prologue then
7:             if scs_reg is loaded in return_blocks then
8:                 shadow_call_stack ← True
9:     shadow_call_stack ← False
```

3.3 Implementation

SeeCFI[4] is purely written in Python in approximately 1300 lines of code and makes use of the angr framework [64]. angr is responsible for loading the binaries, finding the symbols (Algorithm 1), and generating the CFGs (Algorithm 2 and Algorithm 3). SeeCFI aims to detect the deployment of forward- and backward-edge CFI based on LLVM's implementation, which is part of its compiler Clang. The structure of the GCC is checked based on the presence of the function `__VLTVerifyVtablePointer`. We could not identify any binaries compiled with this GCC's CFI implementation. In fact, the option to compile a binary with CFI is unavailable by default in GCC. It is necessary to manually rebuild GCC with the CFI option set to make it available [78]. Therefore, it is rather unlikely that it is widely used.

4 METHODOLOGY

We first explain our data collecting, giving more details about the used images in Section 4.1. Next, we provide our research questions and describe the experimental setup for each of them in Section 4.2.

4.1 Data Collection

In total, we collected 102 different system images: 51 Android images and 51 Linux images. We analyzed about 610K binaries, of which ~91.5% were compiled from a memory-unsafe language (C/C++). The majority of analyzed binaries are 64-bit, and overall, less than 5% were 32-bit-based executables.

4.1.1 Android Images. Our data contains Android versions spanning from Android 7.1.0 to 13.0.0, covering the years from 2017 to 2022. We have one additional Android image of version 4.4, released in 2015. We use the versions released before 2018 (forward-edge CFI was added to Android in 2018) to evaluate the performance of SeeCFI described in Section 5.2. Five out of the 51 Android images originate from Samsung, downloaded from [60], and belong to the Samsung Galaxy S series (see Table 2 in the Appendix). The versions span from Android 8.0.0 to Android 12.0.0. All other images are designed for the Google Pixel devices. Our dataset includes images from the Pixel XL (*marlin*)

[4]https://github.com/software-engineering-and-security/SeeCFI

phone to the Pixel 7 Pro (*cheetah*) device. We analyze 19 images provided by Google [63], ranging from Android versions 7.1.0 to 13.0.0, and depicted in more detail in Table 2 in the Appendix. In addition, we analyze the most recent images of the GrapheneOS [29] operating system, spanning from Android version 12.0.0 to 13.0.0 and available for Pixel 3 to Pixel 7 Pro. Moreover, we study the corresponding Google images [63].

4.1.2 Linux Images. We apply SeeCFI to 51 Linux images in total, 26 Debian and 25 Ubuntu images. The Ubuntu [77] versions range from 12.04.4 to 22.10 of desktop images and cover the years from 2014 to 2022. The Debian [19] versions span from 5.0 to 11.5.0 of live images published between 2009 and 2022. The Linux kernel also uses LLVM's CFI implementation [15]. Table 3 in the Appendix contains more details about the different versions and images.

4.2 Experimental Setup

All experiments analyze binary files extracted from system images and follow the generic process described in Section 4.2.1. It is not necessary to analyze the images more than once as SeeCFI produces deterministic results since it is based on a strict pattern-matching approach. Nevertheless, there are slight variations between different experiment setups aiming at answering the following questions:

RQ1: How does the deployment of forward-edge CFI evolve throughout the different Android and Linux versions based on Google, Samsung, Debian, and Ubuntu releases? (see Section 4.2.2)

RQ2: How much more is forward-edge CFI incorporated into GrapheneOS, a system specialized in security and privacy, compared to Google's releases? (see Section 4.2.3)

RQ3: To what extent is the backward-edge CFI option used in the different Android versions and GrapheneOS, and does that correspond to forward-edge CFI deployment? (see Section 4.2.4)

The experimental setup to evaluate the performance of SeeCFI is described in Section 4.3.

4.2.1 Analyzing System Images. We use SeeCFI to analyze all binaries included in an image file (`.img` or `.iso`). The whole pipeline of this process is illustrated in Figure 2. So far, SeeCFI can analyze Android-based images and Linux images based on Debian, e.g., Ubuntu. In the first step, all binary files are extracted from the image. This can include additional steps to extract binaries contained in special files such as `.apex` files, `.deb` packages, or compressed `squashfs`. These files are unpacked and, if necessary, mounted individually. SeeCFI then iterates over the list of all binaries. The following steps are performed for each binary individually. A database lookup is performed to determine whether the binary already exists. Then, it is checked if the binaries were compiled from C or C++ source code. This is done by a simple `grep` command to scrutinize whether the tag "`.note.gnu.build-id`" is present in the binary. Supposing the binary is marked as compiled from a memory-unsafe language. In that case, it is analyzed by SeeCFI, and the results, including the binary's general information, such as name and path, are added to the database.

Practical Considerations — All files identified as ELF files by the `file` tool are added to our database. If `angr` cannot load a binary file or generate its CFG, SeeCFI returns an error message. This error message and the corresponding binary file are added to the database, indicating that this binary could not be analyzed or could only be partly analyzed. If the binary could not be loaded and thus not analyzed at all, it is not considered in the results below. Most of the errors occur due to an `angr` issue. Moreover, for performance reasons, SeeCFI terminates the CFG generation if it takes longer than 15 minutes. If the CFG cannot be generated, the binary can still be partly analyzed, resulting in flagging it as compiled without forward-edge CFI. Android uses the multi-module option of LLVM's forward-edge CFI implementation [51], SeeCFI checks this without generating

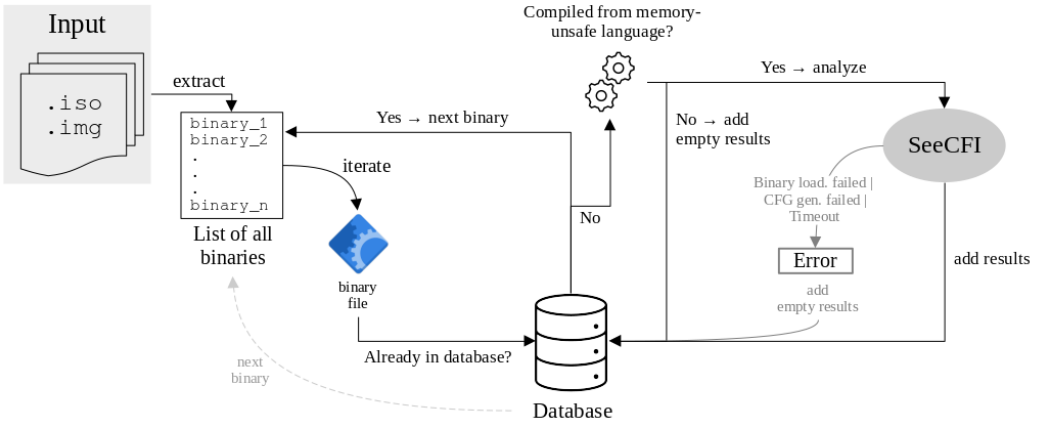

Fig. 2. System image analysis pipeline

the CFG of the given file. Only if it cannot detect multi-module does it generate the CFG and check the presence of single-module CFI. If the generation fails, it still means that the binary was not compiled using the multi-module option. Therefore, it is considered not compiled with forward-edge CFI in general.

In our implementation, we distinguish between three errors that lead to the binary not being analyzed completely. The first occurs when angr cannot load the binary. This means SeeCFI cannot analyze the binary at all. The second and third errors are raised when the CFG cannot be generated. Either because angr could not generate the CFG due to internal errors. Or caused by a timeout error raised by SeeCFI. Among the memory-unsafe binaries analyzed, ~5.5% could not be processed due to one of the previously mentioned errors. Of these failed cases, 73% were due to angr being unable to load the binary, while the remaining 27% occurred because angr could not generate the Control Flow Graph (CFG). No binary caused a timeout.

4.2.2 RQ1 Setup: Android Versions (Google and Samsung) and Linux Versions (Debian and Ubuntu).

Android Versions— To answer the first question, we analyzed 19 different images released by Google and five different Samsung images spanning from Android 7.1.0 to the most recent version, Android 13.0.0. The images used for the experiments are in Table 2 in the Appendix.

We assume that all versions below Android 9 do not have forward-edge CFI as Google only introduced forward-edge CFI by default to the kernel and other components starting with Android 8.1 [52]. For all versions above Android 9, we expect the deployment of forward-edge CFI to increase as security becomes more important for Google over time [23, 53]. Moreover, we expect that Samsung will have a lower adoption rate since they add binaries on top of Google's.

Linux Versions— We analyzed 25 different Ubuntu images and 26 Debian images. The Ubuntu images start with version 12.04 and go up to the latest release of 22.10. The Debian images start with version 5.0 and range up to version 11.5. We only consider amd64-based implementations. For this setup, we also expect to see an increase in forward-edge CFI deployment over time.

4.2.3 RQ2 Setup: GrapheneOS and Google. We analyze 15 factory images released by the open-source project GrapheneOS [29], which is based on the Android operating system and highly focuses on privacy and security. In addition, we analyze the corresponding factory images released by Google to compare the results and to answer the second question. Table 4 in the Appendix shows the phone and Android versions of the tested images. The Pixel series 4 to 7 runs Android

13, and only the Pixel 3 series still runs Android 12. These images correspond to the latest releases by GrapheneOS. Due to its focus on security and privacy, we expect to see a higher deployment of forward-edge CFI in the GrapheneOS releases. GrapheneOS uses a hardened memory allocator[5] to achieve this. It is a custom memory management system designed to enhance security by reducing the risk of memory-related vulnerabilities, such as buffer overflows and use-after-free errors. It achieves this through extensive randomization of memory allocations, isolation of different memory types, delayed freeing of memory to prevent reuse, and using guard pages and consistency checks to detect and prevent heap corruption. This allocator is optimized for security over performance, making it harder for attackers to exploit memory-related weaknesses.

4.2.4 RQ3 Setup: Backward-edge CFI in Android. SeeCFI automatically runs the backward-edge CFI detection on all images mentioned in 4.2.2 and 4.2.3. We evaluate if the deployment of backward-edge CFI corresponds to the deployment of forward-edge CFI, and by that, we can answer the third question. The backward-edge CFI implementation, called ShadowCallStack, is only available for aarch64-based systems due to *time-of-check-to-time-of-use races* [75] security issues on x86_64 systems. Thus, we only consider the Android-based images as an implementation for the collected x86_x64-based Linux images does currently not exist. We expect that the deployment will increase throughout the different Android versions. Moreover, we expect the overall deployment of backward-edge CFI to be higher than forward-edge CFI since backward-edge CFI is easier to implement.

4.3 Performance Evaluation Setup

4.3.1 Makefile and Blueprint Analysis. We evaluate the performance of SeeCFI based on a Makefile and Blueprint analysis of the Android Open Source Project (AOSP) to assess the false positive and false negative rates of SeeCFI. Makefiles and Blueprints are the file types used by the build systems deployed by Android. These files contain compilation specifications, including whether a binary should be compiled using CFI. Makefiles are part of the GNU Project [57] and, for example, used by operating systems such as Debian and Ubuntu. From Android 7 on, Android started moving to the Soong Build System [6], which uses Blueprints. Since then, Android has started to increasingly use Blueprints, but even in the newest versions, there are still remains from GNU Make. In order to analyze them, it is necessary to have access to the source code. This approach is, therefore, not applicable to any vendor image, such as Samsung or Google Pixel devices.

We downloaded the source code of five different Android (AOSP) versions: 4.1, 8.1, 9, 12, and 13. We then parse the included Makefiles and Blueprints to extract which binaries are supposedly compiled with the CFI option. This information is stored as a simple tag [52] within the file. In addition, we have to consider three different files that represent a specific part of the hierarchy of Android's build system. The `cfi_common.mk` file must be parsed, as it contains a list of paths, which causes all files compiled within these paths to use CFI by default. In contrast, the files `cfi_blocklist.txt` and `cfi_blacklist.txt` include lists of paths, files, and functions that are to be excluded from compilation with CFI. Afterward, we compile the source code and leverage SeeCFI on the resulting images. In the last step, we compare the outcome of the Makefile/Blueprint analysis with the output of SeeCFI and, in doing so, we can compute the false positive and false negative rates of SeeCFI.

4.3.2 Additional Manual Analysis. Additionally, we conducted some manual analysis of versions known not to have CFI and programs defined to have CFI enabled.

Assessment of False Positives— We use Android images released before forward- and backward-edge CFI was officially added to Android to verify that the detection does not generate any *false*

[5] https://grapheneos.org/features#hardened-memory-allocator

positive, as these images should result in no binary compiled with CFI. This includes all Android versions before Android 8.1 (2018) (≈8K binaries). Furthermore, we analyzed Debian and Ubuntu images released and updated the last time before 2015 (≈81K binaries), meaning before forward- and backward-edge CFI was implemented in Clang. If SeeCFI returns any binary to be compiled with CFI (≈100 binaries), we inspect it manually using Ghidra [48]. We use the old Android versions to evaluate SeeCFI performance regarding multi-module CFI detection and backward-edge CFI, and the older Linux releases to assess its performance regarding single-module CFI.

Assessment of False Negatives— The second stage is to verify that all the binaries determined to be not compiled with forward- and backward-edge CFI are actually not compiled using forward- and backward-edge CFI. We also achieve this by manually analyzing a subset of binaries using Ghidra and verifying that there is no indication of CFI. To do so, we leverage SeeCFI on the current Firefox ERS binary (version 102.11.0 [45]) to verify that SeeCFI detects forward-edge CFI when it is known to be present. Moreover, we compiled 15 files manually with forward-edge CFI enabled. Then, we manually check if SeeCFI provides the correct results.

5 RESULTS OF SEECFI

In this section, we describe and discuss our results. We start with our observations based on the outcome of the experiments described in Section 4.2. Then we follow with the evolution of forward- and backward-edge CFI. Lastly, we discuss our observations. On average SeeCFI needed 1.5 minutes to analyze each binary, resulting in an overall runtime of, for instance, 45.8 hours in the case of Android 13 (Pixel 7 Pro).

5.1 Observation of Experiments

5.1.1 RQ1: Android (Google and Samsung) and Linux (Debian and Ubuntu) Versions. In this section, we answer the first question and explain, based on our results, how the forward-edge CFI deployment evolves throughout different versions of Android and Linux.

Android Versions— Figure 3 shows the evolution of the number of binaries compiled with forward-edge CFI throughout different Android versions. The introduction of forward-edge CFI started slowly with Android Oreo (Android 8.1) in 2018. In Android 10 (2019), the number of CFI compiled binaries almost tripled from ∼2.7% to ∼7%. It increases even more drastically in Android 11 (2020) by more than 200%, resulting in ∼20.1%. After this enormous spike, the increase slows down slightly. Nevertheless, CFI continues to rise steadily from the ∼20.1% to ∼26.8% in Android 12 (2021) and up to ∼28.7% in Android 13 (2022). This fits our expectation that the forward-edge CFI deployment increases throughout the different Android versions.

Figure 4 displays the total number of binaries analyzed. The number of binaries analyzed is significantly lower in Android 11 (2020) than in Android 12 (2021). Structural changes in the Android ecosystem could explain this decrease in the total number of binaries. Most of the missing files are hardware-related binaries and libraries. In addition, Android 12 introduces a considerable number of new hardware-related binaries that were not present in Android 11 [22]. The increase in forward-edge CFI-compiled binaries can be explained by the fact that Android 13 (2022) focuses more on security than its predecessors [34].

The forward-edge CFI deployment for the Samsung images is almost identical for Android versions 9 (2018) and 10 (2019). For Android versions 11 (2020) and 12 (2021), we can see a significant difference between the adoption of forward-edge CFI in Google and Samsung releases. The similarities in Android 9 and 10 can be explained by the currently, still generally low deployment level (<5%). The difference, on the other hand, is explainable by the fact that Samsung adds further binaries on top of Google's. On average, Samsung has 33% additional binaries, and most of these

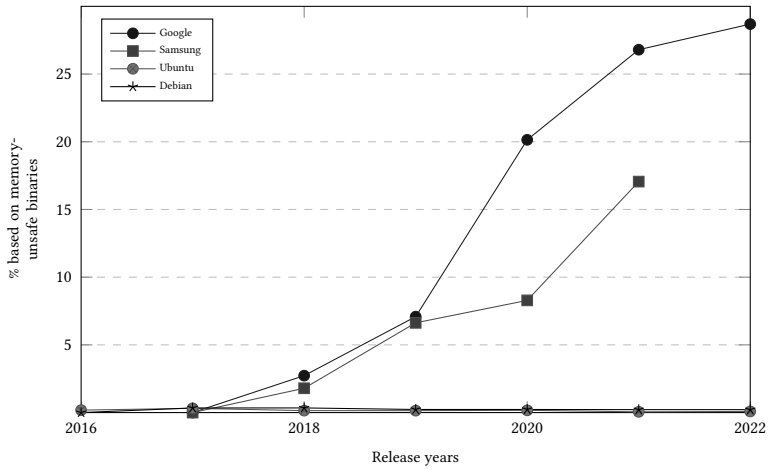

Fig. 3. Evolution of the forward-edge CFI deployment throughout Android versions (Google & Samsung) and Linux versions (Debian & Ubuntu) see Table 1 in the Appendix

binaries do not have forward-edge CFI. This aligns with our expectations that the deployment of forward-edge CFI is lower in Samsung releases.

Overall, the percentage of forward-edge CFI deployment is improving. However, the highest level remains relatively low (~28.7%) (Google), considering the impact of a successful memory corruption attack on a system.

> *Take-away message 1:* Our results show that the deployment of forward-edge CFI is continuously increasing throughout the different Android versions, which indicates that the security is also improving. However, the overall percentage of binaries compiled with forward-edge CFI remains considerably low (~28.7%) when considering how severe a successful memory corruption attack on a system can be. As expected, the deployment of forward-edge CFI is lower in the Samsung images than in the Google releases. Our results suggest that Samsung is not adding any CFI mitigations in their additional binaries.

Linux Versions— As depicted in Figure 3, the deployment in both Linux systems, Debian and Ubuntu, is extremely low (<1%) in comparison to Android. Reasons may be found for the higher total number of binaries in Linux systems than in Android systems. This is illustrated in Figure 4 based on all binaries identified as compiled from a memory-unsafe language, showing that the Linux images have a significantly higher amount than the Android images. The clearly larger quantity of memory-unsafe binaries indicates that the attack surface is substantially more extensive than in Android. The deployment rate of forward-edge CFI in Debian (~0.22% in 2022) is twice as high as in Ubuntu (~0.08% in 2022). However, it is still clear that the difference between Android and Linux deployment rates is significant, leaving the analyzed Linux systems more vulnerable to control-flow hijacking attacks. Moreover, deployment is not increasing, but it is approximately staying at the same level. Our results do not match our expectations. We expected that the deployment would increase, but that is not the case. In addition, we expected the percentage to be generally higher. The binaries containing CFI are primarily part of the Linux kernel [15]. This also explains why Debian and Ubuntu have the same binaries compiled with CFI for their responding versions. We also

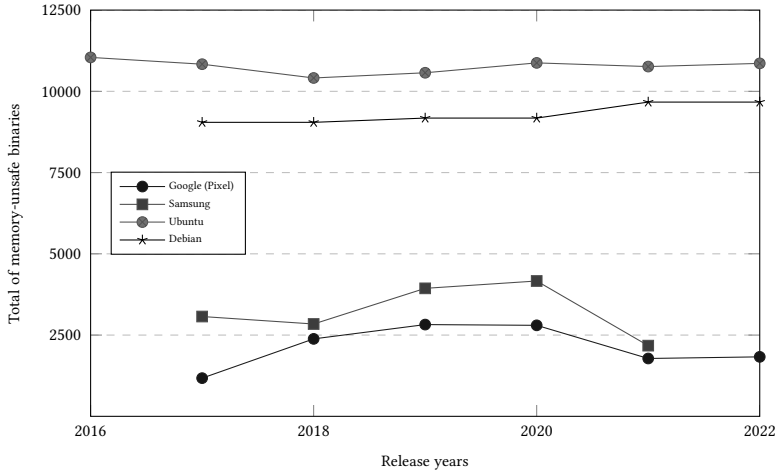

Fig. 4. Comparison of the total amount of memory-unsafe binaries found in the different images of Figure 3

checked for binaries compiled with GCC's CFI option, but SeeCFI could not identify any. Binaries cannot be compiled with CFI when using the standard version of GCC with which operating systems are shipped. Instead, GCC must be rebuilt manually, and thus, it is unlikely that many binaries enable CFI through GCC.

Take-away message 2: The deployment of forward-edge CFI is severely lower in Linux than in Android systems. Moreover, the adoption of forward-edge CFI has not increased throughout the years, indicating that security in this context is not improving. This is especially interesting since the Linux system has, on average, more than five times the amount of memory-unsafe binaries. This leads to the conclusion that Debian and Ubuntu offer a larger attacker surface and are more susceptible to memory corruption vulnerabilities compared to Android. Android uses the multi-module CFI option, while Linux deploys single-module CFI. As Google has proven, it is possible to increase the adoption of CFI to at least ~28.7%, so there is still space for improvement for Debian and Ubuntu.

5.1.2 RQ2: GrapheneOS and Google. In this section, we explain our results regarding question two, and compare the deployment of forward-edge CFI in Google and GrapheneOS releases. Thus, we compare the results of the 15 latest Google and the 15 latest GrapheneOS releases for the different Pixel devices as illustrated in Table 4 in the Appendix. For each GrapheneOS release, we analyze the corresponding Google image. Google and GrapheneOS releases for Pixel phones 4 to 6a are running Android 13, while the releases for the Pixel 3 series use Android 12.

We expected the results to be higher for the GrapheneOS releases compared to Google releases. However, the results in Figure 5 show that GrapheneOS and Google releases have an almost equal adoption rate for the Pixel 3, 6, and 7 series. However, Google has a clearly higher rate for the Pixel 4 and 5 phones. This difference could be explained by GrapheneOS adding binaries on top of Google's. Additionally, although GrapheneOS has a more significant number of CFI-enabled binaries, it still has a lower rate than Google.

There is a slight decrease in binaries compiled with forward-edge CFI from the Pixel 5 series to the Pixel 6 series for Google. This can be explained by the fact that not all features and structural

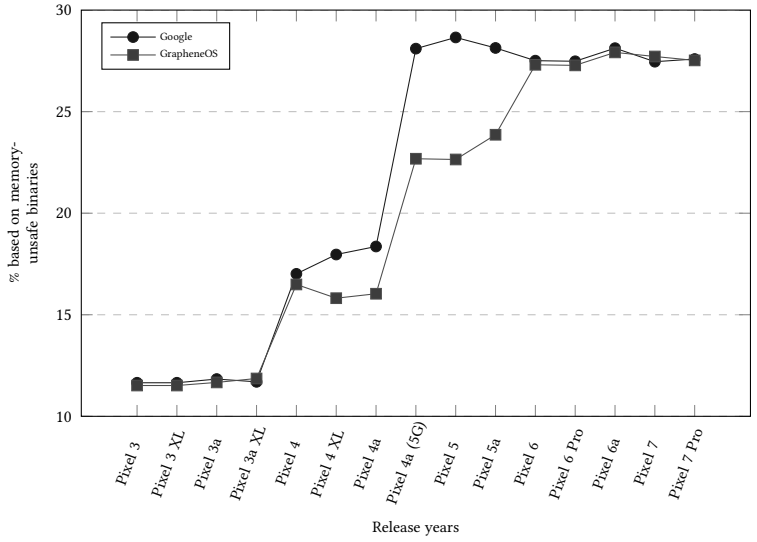

Fig. 5. Evolution of the forward-edge CFI deployment comparing GrapheneOS and Google releases

changes of Android 13 are backported to older phones. Most binaries that cause the drop in the total number of analyzed binaries are related to hardware and kernel modules that are not present in the Android 13 version of the Pixel 6 series anymore [34, 55]. Many of these binaries were compiled using forward-edge CFI, explaining why these structural changes influence the percentage. The additional security and privacy features [28], present in GrapheneOS, could be the reason for the slightly higher number of binaries in their images. GrapheneOS adds ~16.7% to the Pixel 3 series, ~14.2% of binaries to the Pixel 4 and 5 series, and <1% to the Pixel 6 and 7 series. Out of the additional binaries ~14.5% (Pixel 3 series), ~11.5% (Pixel 4 and 5 series), and 0% (Pixel 6 and 7 series) are compiled using forward-edge CFI. GrapheneOS reuses about 84-99.5% of Google's binaries, with the highest reuse rate for Android 13. This explains the high similarities between the results.

Take-away message 3: Contrary to our expectations, the deployment of forward-edge CFI in GrapheneOS releases is not higher than in Google releases. This leads us to two conclusions. On the one hand, CFI does not seem to be an aspect GrapheneOS aims to focus its effort on enforcing security. On the other hand, it could imply that forward-edge CFI is so challenging to deploy that not even a security-focused operating system adopts it at a higher level.

5.1.3 RQ3: Backward-edge CFI in Android. This section answers the third question about the extent of backward-edge CFI (ShadowCallStack) deployment in Android versions released by Google, Samsung, and GrapheneOS. We run the detection of backward-edge CFI on 30 different Google releases, 15 GrapheneOS releases, and five Samsung releases to see the deployment of backward-edge CFI over time. In Figure 6, we summarize the results based on years for the different Android versions, including Google and Samsung releases. Additionally, we provide the percentage of binaries with both forward- and backward-edge CFI. Forward-edge CFI was introduced in Android 8.1 (2018) and backward-edge CFI in Android 9 (2019) [38]. Since backward-edge CFI was introduced in 2019, Figure 6 starts in that year.

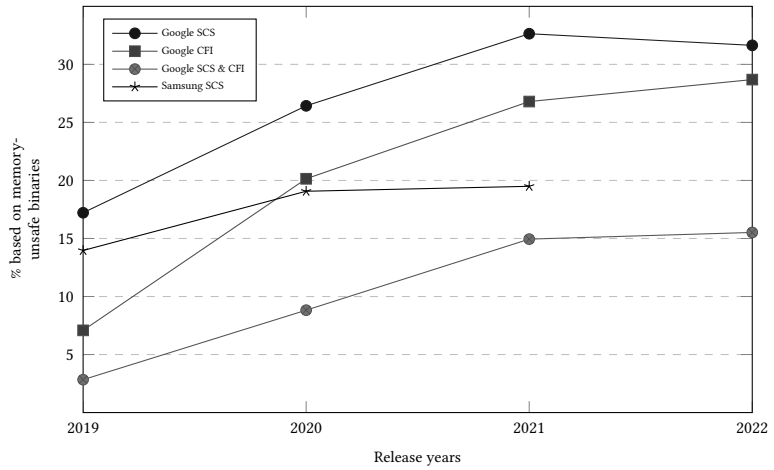

Fig. 6. Evolution of backward-edge CFI deployment throughout different Android versions by Google and Samsung based on release years (see Table 1 in the Appendix)

Figure 6 shows that backward-edge CFI increases more rapidly than forward-edge CFI and has overall a higher deployment throughout all versions. In Android 13, SCS (ShadowCallStack) reaches its highest percentage of ~31.6%. In comparison, ~28.7% of analyzed binaries in the same image are compiled using the forward-edge CFI option. On average, only half of the binaries compiled with forward-edge CFI are also protected by backward-edge CFI (~15.52%). Similar to forward-edge CFI, the adoption of backward-edge CFI in the Samsung releases is proportionally lower than in the Google releases. Figure 7 illustrates the results of comparing the backward-edge CFI

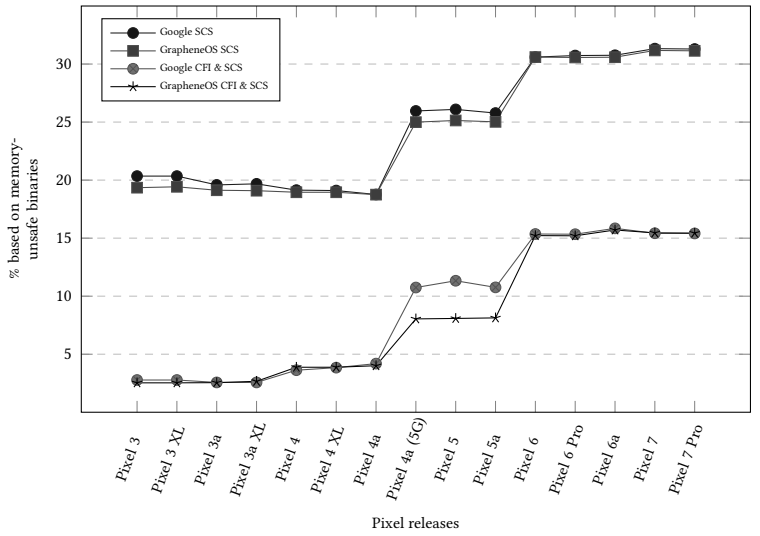

Fig. 7. Evolution of the backward-edge CFI deployment comparing GrapheneOS and Google releases

deployment of GrapheneOS releases with the corresponding Google releases, giving similar results to the comparison of the forward-edge CFI deployment. The adoption in GrapheneOS is almost

identical to the one in Google releases. Only for the Pixel5 series, Google shows a slightly higher deployment. However, the percentage of binaries protected by forward-edge CFI and backward-edge CFI simultaneously is almost identical for Google and GrapheneOS.

Overall, the deployment of the backward-edge CFI option is higher than that of forward-edge CFI. This matches our expectations.

Take-away message 4: Backward-edge CFI is deployed on a higher level than forward-edge CFI for all Android releases. Moreover, our results show that the adoption of SCS is continuously increasing. The higher deployment rate can be explained by the less complex implementation and the lower overhead. For instance, in order to apply forward-edge CFI correctly, the compiler needs to generate the CFG and determine all possible valid destinations of an indirect jump or call. Backward-edge CFI only needs to add storing the return address in the function epilogue on the SCS and popping and comparing it in the function's prologue. This also means the backward-edge control flow (function returns) is better protected than the forward-edge.

5.2 Evaluation of SeeCFI's Performance

5.2.1 Makefile and Blueprint Analysis. The evaluation of the results of the Makefile and Blueprint analysis shows that SeeCFI has a false positive rate of 0 and a false negative rate of 0.17. However, it is interesting to mention that the false negative rate decreases significantly with each incremental Android version. For example, Android 9 showed a FN rate of 0.37, Android 12 merely a rate of 0.015, and Android 13 dropped to 0 false negatives. The high amount of false negatives is caused by binaries that are part of the Android VNDK system. The majority of these binaries are 32-bit, which is one of SeeCFI limitations covered in Section 5.2.3. VNDK is the vendor native development kit and enables Android to separate vendor partitions from the rest of the system, allowing framework-only updates [7]. The structure of these libraries was changed with later Android versions, likely leading to the observed decrease in the false negatives [8]. We manually inspected the VNDK binaries in Android 9 and later versions and observed that about 90% were removed entirely in the later versions. Throughout our performance evaluation, we analyzed a total of 3763 binaries. 533 binaries were correctly classified as compiled with CFI and 2733 without CFI. Both the Makefile/Blueprint analysis and SeeCFI did not detect any binaries compiled with CFI in Android 4 as CFI was not implemented yet. SeeCFI has an average false negative rate of 0.17 and a false positive rate of 0 for its multi-module CFI detection.

5.2.2 Additional Manual Analysis. We first present the results of our manual analysis focusing on false positives, followed by the results of the evaluation of false negatives.

Assessment of False Positives— As expected, the results of Android 4 and 7 show that 0 binaries are compiled with forward-edge CFI. Forward-edge CFI was introduced into Android from version 8.1 onwards. This matches our results and verifies that SeeCFI has no false positives for its multi-module CFI detection. The analysis of the Debian and Ubuntu versions released and updated the last time before 2016 shows that SeeCFI has a false positive rate of 0.12 (110 binaries out of 80770 binaries). We manually inspected the 110 binaries causing this rate. We found some code related to crash handling and forcefully terminating the program[6]. All these FPs occurred in kernel modules. These binaries include precisely the same pattern as forward-edge CFI. Thus, SeeCFI cannot distinguish between them. It is evident that the presence of CFI cannot cause this, as these

[6]An example of code causing false positives can be found at https://github.com/torvalds/linux/blob/v3.11-rc7/fs/coda/file.c.

binaries were compiled before CFI was implemented. SeeCFI has a false positive rate of 0.004 for its single-module CFI detection.

Assessment of False Negatives— We could not find any false negatives for the single-module CFI detection of SeeCFI as it correctly detected all test binaries, including Firefox, as compiled with forward-edge CFI. We additionally chose a random subset of binaries (~50), from the `lib` directories of Ubuntu 22.04.1 and Debian 11.5.0 and manually searched for indications of (single-module) CFI in these. We were not able to identify any. Thus, we conclude that SeeCFI has a false negative rate of 0 for its single-module CFI detection.

5.2.3 Limitations. CFI-unrelated patterns— There still remains the possibility of other CFI-unrelated code patterns resembling the structure of the CFI pattern used for detection. Moreover, we cannot ensure that all binaries compiled with CFI are detected and that none is missed.

Cast checks— Clang's forward-edge CFI implementation includes cast checks, which are not covered by our detection algorithms. However, the evaluation of our Makefile and Blueprint analysis (see 5.2.1) shows that this does not seem to have any impact on our results, as we do not have any false positives.

`x18` *register*— In most cases, Android ensures that the `x18` register, needed as a special purpose register by the backward-edge CFI implementation, is protected and not used by any other instruction. However, this cannot be ensured in specific cases, e.g., when using SP-HALs. The hardware abstraction layer (HAL) is a standard interface used by hardware vendors, allowing Android to be independent of lower-level driver implementations [5]. They are implemented in the form of shared libraries. Same-process HALs (SP-HALs) are "*always open[ed] in the same process in which they are used*" [4].

32-bit binaries— Our evaluation in Section 5.2.1 showed that SeeCFI performs slightly worse for 32-bit binaries. Indeed, only one 64-bit binary resulted in a false negative in all Android binaries analyzed.

Optimization & obfuscation— We do not consider any specific optimization or obfuscation options. Nevertheless, the false positive rate evaluated on Android is 0, and the false negative rate is 0.17.

5.3 Discussion of Experiment Results

In the following, we discuss the observations from Section 5.1 and provide possible explanations for our results. We have three main observations:

(1) The deployment of forward-edge CFI is considerably higher in Android than in Linux systems.
(2) Our results show that the deployment of forward-edge CFI is still relatively low, even though it was introduced more than ten years ago, officially implemented about seven years ago, and officially included in Android's kernel and userspace five years ago. In particular, the deployment is surprisingly poor when considering the severe effects a successful memory corruption attack can have.
(3) The deployment of backward-edge CFI (ShadowCallStack) is higher than the forward-edge CFI adoption.

Adding CFI support to Android came with different challenges. When using LLVM's forward-edge CFI, link-time optimization (LTO) is required. This introduces an overhead as linking requires substantially higher memory and CPU cost [38]. Furthermore, Android's developer team needed to extend and patch the original implementation to make it work without causing programs to crash [38]. This kind of general implementation, including specific fixes, is unavailable for all Linux distributions. kCFI [44] was one approach to adding forward-edge CFI to the Linux kernel. However, their repository [26] was not updated in the last three years, indicating that it is no longer

maintained. Moreover, in Linux systems, it is more of the responsibility of the different developers to enable CFI for their applications than system-wide support [41]. The NX bit, for instance, is used in Linux systems and can be deployed at a system-wide level, requiring little change to the application side. This might lead to CFI's significantly lower deployment rate in Linux systems compared to Android.

The general lack of deployment could have several reasons. The first reason is the still prevalent overhead in performance and size when a file is compiled with CFI, as mentioned before, related to LTO. The Clang team has observed up to 15% overhead with the Chromium browser as the "*scheme has not yet been optimized for [this] binary size*" [70]. The second reason is that forward-edge CFI can cause *function type mismatch errors* and *assembly code type mismatch errors*, which cause the affected program to crash. The latter error is caused by functions calling assembly code directly [52]. They can be fixed by adding these specific functions to a forward-edge CFI blacklist. Considering a high amount of functions calling assembly code, the maintenance of such a blacklist would have a negative impact on the development time. Moreover, even though CFI reduces the attack surface and thus makes the exploitation of memory corruption vulnerabilities more complex, significant ongoing research has discovered ways to bypass forward- and backward-edge CFI. We give an overview of possible bypasses in Section 6. All these are reasons not to deploy or only deploy CFI to a limited extent.

The fact that backward-edge is deployed on a higher level than forward-edge could be caused by the possible overhead introduced by forward-edge CFI mentioned in Section 5.1.3. The higher utilization rate results from a less complex implementation and lower overhead. For example, to use forward-edge CFI correctly, the compiler must generate the CFG and determine all possible valid targets of an indirect jump or call. Backward-edge CFI, in contrast, only needs to store the return address on the shadow stack. This is done in the function's prologue. This value is then restored from the shadow stack and compared with the return address obtained from the *regular* stack. These instructions are performed in the function's epilogue.

In order to evaluate future implications, we consider Android. Although CFI was added to the Android system roughly five years ago, memory corruption vulnerabilities still pose the majority of severe Android vulnerabilities in 2022 [43]. This caused Android to adopt additional memory protection mechanisms, such as Memory Tagging Extensions (MTE) [9], in the future. This leads us to the conclusion that in the future, the use of hardware support will be needed, such as MTE, which is "*a hardware implementation of tagged memory*" [9]. However, the long-term solution will presumably be to switch more and more to memory-safe languages, as Android has started to do. For instance, Android developers started moving from writing low-level code in memory-unsafe languages such as C and C++ to the memory-safe language [65, 66] Rust [69], eliminating the need for CFI completely [42]. Although Rust comes with its own challenges, it continues to fulfill its promises regarding a safer memory, as shown by Xu et al. [79]. They studied all known Rust-related CVEs, 186 at the time of 2020-12-31. They found that they all relate to "*including automatic memory reclaim, unsound function, and unsound generic or trait*" [79].

Since it is unlikely that most of the existing code will be rewritten in memory-safe languages[7], CFI, in combination with other techniques, is still one of the most promising mitigation mechanisms if it is used. In essence, while the other protections (stack canaries, FORTIFY, NX, ASLR, and PIE) make it harder to exploit memory vulnerabilities, CFI ensures explicitly that even if an attacker can

[7]"*Of course, introducing a new programming language does nothing to address bugs in our existing C/C++ code. Even if we redirected the efforts of every software engineer on the Android team, rewriting tens of millions of lines of code is simply not feasible.*" [66]

corrupt memory, they cannot easily redirect the program's execution to their malicious code. This makes the overall security of the binary much more robust.

5.4 Additional Mitigation Techniques

We checked the presence of four standard mitigation techniques against memory corruption vulnerabilities that should be used in combination with CFI. Each image and, if needed, all included binaries were checked individually.

5.4.1 Address Space Layout Randomization (ASLR) and Position-Independent Executable (PIE). are both techniques aimed at increasing the security of a system by making it more difficult for an attacker to predict the memory locations of critical components. ASLR works by randomly arranging the address space positions of key data areas such as the stack, heap, and libraries, making it harder for attackers to exploit memory corruption vulnerabilities. PIE complements ASLR by enabling the entire executable to be loaded at random memory addresses rather than at a fixed location, further enhancing its effectiveness. We verified if ASLR is enabled by checking the system-wide configuration; see Section 7.2.4. PIE needs to be identified on a binary level by applying the tool `hardening-check`[8]. Our results show that Android (Google, Samsung, and GrapheneOS), Debian, and Ubuntu have both mitigation techniques enabled, system-wide and for each binary.

5.4.2 Data Execution Prevention (DEP) and the NX (No-eXecute). bit, on the other hand, are related to preventing certain types of code execution attacks. DEP is a security feature that marks certain memory areas as non-executable, meaning that the code cannot be executed even if an attacker injects code into these areas. The NX bit is a hardware feature that enforces this protection by flagging specific memory pages as non-executable. DEP is an umbrella term, while the NX bit is a specific hardware implementation. DEP features protect against specific exploits, such as buffer overflows, by ensuring that injected code cannot be executed, even if it is successfully placed in memory, e.g., on the stack. We used `hardening-check` to determine whether the system was protected by preventing data execution. In addition, we verified whether the NX bit was set at the system level. All considered operating systems are entirely protected by DEP, more precisely by its No-eXecutable (NX) hardware implementation.

6 LIMITATIONS OF CONTROL FLOW INTEGRITY

CFI mitigates the ability of attackers to launch control flow attacks. However, it cannot completely prevent these types of attacks, as there are proven ways to bypass CFI. For this reason, researchers are constantly striving to find new vulnerabilities to get ahead of attackers. In this section, we summarize the known possibilities for bypassing forward-edge CFI and backward-edge CFI.

6.1 Forward-edge CFI

One of the most recent attacks is called *WrapAttack*. Xu et al.'s [80] attack is based on the discrepancy that compilers do not distinguish between code introduced for security reasons and application code. This introduces an additional attack vector leading to *Time-Of-Check-to-Time-Of-Use* (TOCTTOU) attacks. Conti et al. [14] describe another attack that benefits from compiler optimization that violates the security assumptions made by the mitigation mechanism. The vulnerability is that security-relevant register values can be spilled on the stack. CFI and other mechanisms assume that the compiler handles these values in a protected way. However, this is not the case in practice.

Göktas et al. [27] show in their paper that it is possible to perform control-flow hijacking attacks still using a specific type of code-reuse attack. They introduced two kinds of gadgets, CS (*call-site*)

[8]https://manpages.ubuntu.com/manpages/bionic/man1/hardening-check.1.html

and EP (*entry-point*), that can be chained together to build a malicious payload. The presence of ASLR makes the attack more sophisticated as the localization of the needed gadgets gets more complex. However, ASLR is prone to data leakage vulnerabilities and can be bypassed. Moreover, they mention in their paper the possibility of accessing a protected sensitive function when being able to locate a gadget containing a direct call to this function. Forward-edge CFI does not protect direct calls; thus, calling them does not violate CFI policies.

Evans et al. [25] describe a similar approach using ACICS (*Argument Corruptible Indirect Call Site*) gadgets. Attackers can use these gadgets to perform a Remote Code Execution. This attack is called *control Jujutsu*. Even though Tice et al.'s paper [76] was published in 2015, Sayeed et al. [61] showed in 2019 that LLVM's and GCC's CFI implementations still fail to protect software against this attack.

Control-Flow Bending is another attacker introduced by Carlini et al. [13] and is similar to the ones mentioned before. Instead of hijacking the control flow, the attacker "bends" the control flow within the allowed CFG, meaning that the attacker only uses legitimate call and jump targets. These attacks are mainly possible due to the over-approximation in the generated CFGs, resulting in the graphs containing more edges (execution paths) than the actual execution. Constructing a sound and complete/precise CFG solely based on static analysis is not possible. However, there is no scalable way of using another approach to gather the information needed to build a sound and precise CFG. An unsound CFG can impede the proper functionality of a program. Therefore, an imprecise CFG contains more execution paths than actually exists. This imprecision increases the attacker's surface by allowing them to exploit the additional call and jump targets. Nevertheless, Evans et al. [25], and Carlini et al. [13] have shown that even a fully precise forward-edge CFI implementation with a sound and precise CFG would be prone to these attacks.

6.2 Backward-edge CFI (ShadowCallStack)

One of the obvious limitations of backward-edge CFI's implementation is that it is only available for aarch64-based systems. When LLVM introduced backward-edge CFI in the form of the *ShadowCallStack* with version 7.0.1, they provided an implementation for x86_64. However, this was removed in version 9.0.0 as it introduced severe security risks [74]. When using backward-edge CFI in x86_64 a *time-of-check-to-time-of-use* (TOCTOU), an attack is possible since x86_64 does not store the return address in a register but on the stack [17, 75]. This means that only aarch64-based systems can be protected by backward-edge CFI as the `ret` and `call` instructions operate on registers.

Conti et al. [14] show in their paper that if it is possible to leak the pointer pointing to the ShadowCallStack, it is possible to tamper with its content. Moreover, SCS does not support a multi-threading solution [83]. Thus, new approaches, such as Bustk [83], need to be researched.

7 RELATED WORK

In order to make forward-edge CFI more secure against the attacks mentioned in Section 6, it needs to be combined with other protection mechanisms such as the ones trying to achieve complete memory safety [25, 47], shadow call stacks [13], or adding runtime information to the construction process of the CFG [27]. All three papers, [13, 25, 27], were written almost a decade ago, raising the question of how secure today's CFI implementations are. In Section 7.1, we summarize papers evaluating the effectiveness of Control Flow Integrity. Afterward, we give an overview of other memory protection mechanisms recommended to use in combination with CFI in Section 7.2.

7.1 Effectiveness of Control Flow Integrity

Muntean et al. [46] created a framework, LLVM-CFI, to evaluate the effectiveness of eight different CFI policies, including the policies used for the adoption in GCC and LLVM, based on Tice et al.'s

paper [76]. They measure the total number of "*calltargets*" after applying the different CFI policies, making it possible to compare them easily. This new metric is called *calltarget reduction* (CTR). Their results have shown that the evaluated policies are too permissive, meaning they still leave space for an attacker to perform control flow hijacking attacks.

Li et al. [33] evaluated the twelve most common open-source implementations of CFI. They introduce two tools, CScan and CBench. The latter verifies the effectiveness of the different implementations against common control-flow hijacking attacks. The former measures the security boundaries of CFI implementations. They first enumerate all potential code addresses and then verify whether the protected instructions are allowed to jump there. Their results show that ten out of twelve contain flaws, such as introducing new unindented jump targets.

Our tool SeeCFI can also be used to check that the toolchain used to generate CFI is working correctly by verifying that CFI is present in the final binary.

7.2 Memory Protection Mechanisms

7.2.1 Data Execution Prevention (DEP). The simplest form of buffer overflows or *stack smashing* consists of writing the code the attacker wants to execute (i.e., the payload) into the buffer. This kind of attack can be prevented by making specific memory segments non-executable. Only the segments containing the code are executable. This means any data and the stack are part of the non-executable memory, as implemented by ExecShield [30]. This is similar to the principle of read-only permissions. This defense mechanism is called Data Execution Prevention or DEP. Nevertheless, attackers can circumvent DEP by not writing their malicious code directly into the buffer but by either reusing code from the program (*code-reuse attack*) or having the pointer point to their own code in a memory zone marked executable. DEP is enabled on all common operating systems by default. This can be easily verified by checking the `NX/XD` bit– also known as *No-eXecute*.

7.2.2 Stack canaries. or stack cookies are a value added after a vulnerable buffer on the stack to prevent attacks from overwriting the return address pointer. The value of the stack canary is checked before jumping back using the return address pointer so that a buffer overflow attack can be detected, assuming that the attacker does not know the value. If the stack canary check fails, the program usually terminates or crashes. StackGuard is a compiler extension implementing three different kinds of stack canaries [16]: (i) *random canaries*, (ii) *terminator canaries*, and *random XOR canaries*. For each execution, a new value is generated. Different implementations and integrations exist, for instance, Microsoft Windows implementation /GC [36]. However, since stack canaries are only safe, assuming that an attacker does not know the value, they can be bypassed by an adversary with this knowledge. The attacker could use brute-forcing or a data leakage vulnerability (e.g., a format string vulnerability). Moreover, it cannot prevent heap-based buffer overflows. The presence of stack canaries in a binary can be identified in the function's epilogue as the canary check is inserted there.

7.2.3 FORTIFY. is a security feature that enhances the safety of standard C library functions like `strcpy`, `memcpy`, and `sprintf` [56], which are prone to buffer overflows. It works by automatically replacing these vulnerable functions with safer versions that perform additional checks at compile time and runtime. If FORTIFY detects that a buffer might be overrun, it either stops the execution or logs a warning, depending on the severity, thus preventing many common memory corruption vulnerabilities. It is utilized by major operating systems such as Android [10].

7.2.4 Address-Space Layout Randomization - ASLR. mitigates code-reuse attacks by introducing randomness into addresses of code segments [50], which means that their memory addresses are unpredictable. In order to be able to reuse a specific code part or function of the current program, the

attacker needs to know the memory address of these to overwrite the return address pointer with this exact address. A more sophisticated code-reuse attack method is called *return-oriented programming* (ROP). An attacker chains sequences of instructions, so-called gadgets, only reusing existing code from the program. The attacker needs to know the exact locations of these gadgets. Therefore, ASLR can also prevent this attack. However, ASLR can be bypassed by the same techniques as stack canaries. ASLR is enabled by default in the kernel (2.6.16 and higher). For instance, in Linux, it is possible to manually check the `randomize_va_space` file in `/proc/sys/kernel`.

7.2.5 Position-Independent Executable - PIE. is a security feature that allows an executable to be loaded at random memory addresses each time it is run rather than at a fixed address. This randomness enhances security by making it significantly more difficult for attackers to predict where specific code or data is located in memory, a common technique for exploiting vulnerabilities. PIE enables the entire executable, including its code, to be position-independent, similar to how shared libraries function. When combined with Address Space Layout Randomization, PIE significantly strengthens a system's defenses against specific attacks, such as buffer overflows and Return-Oriented Programming (ROP). PIE is supported by many modern operating systems, including Linux, such as Debian [18] and Ubuntu [1], and Android [21], and is often enabled by default in the compilation process for critical software.

7.2.6 Pointer Authentication Code - PAC. is an ARMv8.3 extension [58, 59]. In AARCH64, only the last 40 bits of a pointer are used, which leaves the most significant 24 bits for other purposes. Pointer Authentication uses this fact by adding a signature (= PAC) into these bits [59, 67, 72]. This signature is verified before using the pointer. If the verification fails, the program terminates or crashes. However, PAC is not completely secure against code-reuse attacks [82]. Apple uses Pointer Authentication Codes (PAC) [67].

8 CONCLUSION & FUTURE WORK

We have shown that our tool SeeCFI can determine whether a binary was compiled using forward- and/or backward-edge CFI. Furthermore, it can differentiate between single-module and multi-module CFI. In total, we applied SeeCFI to 102 different system images, 51 Android and 51 Linux images, to identify forward- and backward-edge CFI deployment throughout versions and times. We considered the Samsung, Google, and GrapheneOS releases focusing on privacy and security. The deployment of backward-edge CFI is generally higher than forward-edge CFI. We then discussed the different reasons explaining the results of our experiments.

In summary, the adoption of forward-edge CFI is not significantly increasing (always <1%) within Linux (Debian & Ubuntu) systems, indicating that it is unlikely to happen in the future. In Android, on the other hand, the deployment increases over time, ending at ~28.7% for Android 13. However, Android started moving to memory-safe languages, making CFI redundant in the context of memory-safe languages generating native code. Nevertheless, existing code written in memory-unsafe languages will still rely on CFI's protection in the future.

SeeCFI enables determining which binaries have been compiled in a system with CFI. This can be helpful for system developers to decide which binaries to focus on first when fixing vulnerabilities, as binaries without CFI make a better target. This assumption is reasonable as we have explained (Section 2 and Section 6) how CFI can significantly increase the complexity of possible attacks.

In the future, we plan to add the detection of Clang's Pointer Authentication Code used by Apple and Windows' CFI implementation called Control Flow Guard to SeeCFI.

REFERENCES

[1] 0xnishit. 2024. Security/Features - Ubuntu Wiki. https://wiki.ubuntu.com/Security/Features. Accessed 2024-10-02.

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.

[3] Starr Andersen and Vincent Abella. 2004. Memory protection technologies. *Changes to Functionality in Windows XP Service Pack* 2 (2004).

[4] Android Open source Project. 2023. Hardware abstraction layer overview | Android Open Source Project. https://source.android.com/docs/core/architecture/hal. Accessed 2023-12-01.

[5] Android Open source Project. 2023. Legacy HALs | Android Open Source Project. https://source.android.com/docs/core/architecture/hal/archive. Accessed 2023-12-01.

[6] Android Open source Project. 2023. Soong Build System | Android Open source Project. https://source.android.com/docs/setup/build. Accessed 2023-11-17.

[7] Android Open source Project. 2023. Vendor Native Development Kit (VNDK) overview | Android Open Source Project. https://source.android.com/docs/core/architecture/vndk. Accessed 2023-12-01.

[8] Android Open source Project. 2023. VNDK Build System Support | Android Open Source Project. https://source.android.com/docs/core/architecture/vndk/build-system. Accessed 2023-12-01.

[9] AOSP. 2023. Arm Memory Tagging Extension | Android Open Source Project. https://source.android.com/docs/security/test/memory-safety/arm-mte. Accessed 2023-06-05.

[10] George Burgess. 2017. Android Developers Blog: FORTIFY in Android. https://android-developers.googleblog.com/2017/04/fortify-in-android.html. Accessed 2024-10-02.

[11] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 985–999.

[12] Davide Capodaglio. 2024. Fixed/defualt attributes from XSD are saved as normal attributes (#691)... https://gitlab.gnome.org/GNOME/libxml2/-/issues/691. Accessed 2024-09-30.

[13] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. {Control-Flow} Bending: On the Effectiveness of {Control-Flow} Integrity. In *24th USENIX Security Symposium (USENIX Security 15)*. 161–176.

[14] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. 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*. 952–963.

[15] Jonathan Corbet. 2021. Control-flow integrity in 5.13 [LWN.net]. https://lwn.net/Articles/856514/. Accessed 2023-12-05.

[16] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX security symposium*, Vol. 98. San Antonio, TX, 63–78.

[17] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 555–566.

[18] Debian. 2022. Hardening - Debian Wiki. https://wiki.debian.org/Hardening#gcc_-pie_-fPIE. Accessed 2024-10-02.

[19] debian.org. 2023. Index of /mirror/cdimage/archive. https://cdimage.debian.org/mirror/cdimage/archive/. Accessed 2023-03-15.

[20] Solar Designer. 1997. Getting around non-executable stack (and fix). *http://ouah. bsdjeunz. org/solarretlibc. html* (1997).

[21] Android Developers. 2017. NDK Revision History | Android NDK | Android Developers. https://developer.android.com/ndk/downloads/revision_history. Accessed 2024-10-02.

[22] Android Developers. 2022. Android 12 features and change list | Android Developers. https://developer.android.com/about/versions/12/summary. Accessed 2023-12-05.

[23] Android Developers. 2022. Security tips | Android Developers. https://developer.android.com/training/articles/security-tips. Accessed 2022-10-04.

[24] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding integer overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 1–29.

[25] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 901–913.

[26] github kCFI. 2023. kcfi/kcfi: Control-Flow Integrity implementation for the Linux Kernel 3.19. https://github.com/kcfi/kcfi. Accessed 2023-06-05.

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

[28] GrapheneOS. 2022. Feature overview | GrapheneOS. https://grapheneos.org/features. Accessed 2022-08-31.

[29] GrapheneOS. 2022. GrapheneOS: the private and secure mobile OS. https://grapheneos.org/. Accessed 2022-08-31.

[30] Red Hat Blog Huzaifa Sidhpurwala. 2018. Security Technologies: ExecShield. https://www.redhat.com/en/blog/security-technologies-execshield. 2018-07-25.

[31] Free Software Foundation Inc. 2022. Instrumentation Options (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gcc/Instrumentation-Options.html. Accessed 2022-07-06.

[32] Henrich Lauko, Lukáš Korenčik, and Petr Ročkai. 2022. Verification of programs sensitive to heap layout. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–27.

[33] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. 2020. 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*. 1821–1835.

[34] Eugene Liderman and Sara N-Marandi. 2022. Google Online Security Blog: I/O 2022: Android 13 security and privacy (and more!) . https://security.googleblog.com/2022/05/io-2022-android-13-security-and-privacy.html. Accessed 2023-12-05.

[35] Jin Lin. 2021. Developer Guidance for Hardware-enforced Stack Protection - Microsoft Tech Community. https://techcommunity.microsoft.com/t5/windows-kernel-internals-blog/developer-guidance-for-hardware-enforced-stack-protection/ba-p/2163340.

[36] Microsoft. 2021. \GS (Buffer Security Check) | Microsoft Docs. https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check?view=msvc-170.

[37] Microsoft. 2022. Control Flow Guard - Win32 apps | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[38] Jake Edge | LWN.net. 2023. Control-flow integrity for the kernel [LWN.net]. https://lwn.net/Articles/810077/. Published 2020-01-22 (Accessed 2023-05-31).

[39] NIST | National Institute of Standards and Technology. 2022. NVD - CVE-2022-20127. https://nvd.nist.gov/vuln/detail/CVE-2022-20127. Accessed 2022-10-04.

[40] NIST | National Institute of Standards and Technology. 2023. NVD -CVE-2023-2731. https://nvd.nist.gov/vuln/detail/CVE-2023-2731. Accessed 2024-09-30.

[41] Marco Benatto | RedHat. 2023. Fighting exploits with Control-Flow Integrity (CFI) in Clang. https://www.redhat.com/en/blog/fighting-exploits-control-flow-integrity-cfi-clang. Published 2020-05-26 (Accessed 2023-05-31).

[42] Aamir Siddiqui | XDA. 2023. Google is developing parts of Android in Rust to improve security. https://www.xda-developers.com/google-developing-android-rust/. Published 2021-04-06 (Accessed 2023-05-31).

[43] Mishaal Rahman | XDA. 2023. Android 14 may add an advanced memory protection feature to protect your device from memory safety bugs. https://www.xda-developers.com/android-14-advanced-memory-protection/. Accessed 2023-05-31.

[44] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. DROP THE ROP fine-grained control-flow integrity for the Linux kernel. *Black Hat Asia* 2017 (2017).

[45] moz:\\a. 2023. Firefox ERS 102.11.0, See All New Features, Updates and Fixes. https://www.mozilla.org/en-US/firefox/102.11.0/releasenotes/. Accessed 2023-06-04.

[46] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2019. Analyzing control flow integrity with LLVM-CFI. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 584–597.

[47] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 2010 international symposium on Memory management*. 31–40.

[48] NationalSecurityAgency. 2022. GitHub - NationalSecurityAgency/ghidra: Ghidra is a software reverse engineering (SRE) framework. https://github.com/NationalSecurityAgency/ghidra. Accessed 2022-08-28.

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

[50] PaXTeam. 2003. pax.grsecurity.net/docs/aslr.txt. https://pax.grsecurity.net/docs/aslr.txt.

[51] Android Open Source Project. 2020. Control Flow Integrity | Android Open Source Project. https://source.android.com/devices/tech/debug/cfi.

[52] Android Open Source Project. 2022. Control Flow Integrity | Android Open Source Project. https://source.android.com/docs/security/test/cfi. Accessed 2022-10-04.

[53] Android Open Source Project. 2022. Security and Privacy Enhancements in Android 10 | Android Open Source Project. https://source.android.com/docs/security/enhancements/enhancements10. Accessed 2022-10-04.

[54] Android Open Source Project. 2022. ShadowCallStack | Android Open Source Project. https://source.android.com/devices/tech/debug/shadow-call-stack.

[55] Android Open Source Project. 2023. Kernel modules overview | Android Open Source Project. https://source.android.com/docs/core/architecture/kernel/modules. Accessed 2023-12-05.

[56] The GNU Project. [n. d.]. Source Fortification (The GNU C Library). https://www.gnu.org/software/libc/manual/html_node/Source-Fortification.html. Accessed 2024-10-02.

[57] GNU Project supported by Free Software Foundation. 2023. Make - GNU Project - Free Software Foundation. https://www.gnu.org/software/make/. Accessed 2023-11-17.

[58] Inc.) Qualcomm Product Security (Qualcomm Technologies. 2017. Pointer Authentication on ARMv8.3 | Design and Analysis of the New Software Security Instructions. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[59] Mark Rutland. 2017. Pointer authentication in AArch64 Linux – The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/arm64/pointer-authentication.html.

[60] SamMobile. 2023. SamMobile - Your source for all Samsung news. https://www.sammobile.com/. Accessed 2023-04-26.

[61] Sarwar Sayeed, Hector Marco-Gisbert, Ismael Ripoll, and Miriam Birch. 2019. Control-flow integrity: attacks and protections. *Applied Sciences* 9, 20 (2019), 4229.

[62] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 309–318.

[63] Google Play services. 2023. Factory Images for Nexus and Pixel Devices | Google Play services | Google for Developers. https://developers.google.com/android/images. Accessed 2023-04-10.

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

[65] Jeffrey Vander Stoep. 2022. Google Online Security Blog: Memory Safe Languages in Android 13 . https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html. Accessed 2023-12-05.

[66] Jeff Vander Stoep and Android Team Stephen Hines. 2021. Google Online Security Blog: Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html. Accessed 2024-10-02.

[67] Apple Support. 2021. Operating system security - Apple Support. https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/1/web/1#sec0167b469d.

[68] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[69] Rust Team. 2023. Rust Programming Language. https://www.rust-lang.org/. Accessed 2023-06-05.

[70] The Clang Team. 2022. Control Flow Integrity – Clang 15.0.0.git documentation. https://clang.llvm.org/docs/ControlFlowIntegrity.html. Accessed 2022-07-05.

[71] The Clang Team. 2022. Control Flow Integrity Design Documentation – Clang 15.0.0.git documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html. Accessed 2022-07-05.

[72] The Clang Team. 2022. Pointer Authentication – Clang 15.0.0.git documentation. https://llvm.org/docs/PointerAuth.html#armv8-3-a-pauth-pointer-authentication-code. Accessed 2022-07-06.

[73] The Clang Team. 2022. ShadowCallStack – Clang 15.0.0.git documentation. https://clang.llvm.org/docs/ControlFlowIntegrity.html. Accessed 2022-07-06.

[74] The Clang Team. 2022. ShadowCallStack - Clang 16.0.0git documentation. ttps://clang.llvm.org/docs/ShadowCallStack.html. Accessed 2022-09-20.

[75] The Clang Team. 2022. ShadowCallStack - Clang 7 documentation. https://releases.llvm.org/7.0.1/tools/clang/docs/ShadowCallStack.html. Accessed 2022-09-20.

[76] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX security symposium (USENIX security 14)*. 941–955.

[77] ubuntu.com. 2023. Index of /releases. https://old-releases.ubuntu.com/releases/. Accessed 2023-03-15.

[78] Jonathan Wakely. 2014. vtv - GCC Wiki. https://gcc.gnu.org/wiki/vtv. Accessed 2023-12-05.

[79] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. 2021. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–25.

[80] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. 2023. WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1271–1288.

[81] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2003. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.* IEEE, 260–269.

[82] Brandon Azad (Project Zero). 2019. Project Zero: Examining Pointer Authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[83] Changwei Zou, Xudong Wang, Yaoqing Gao, and Jingling Xue. 2022. Buddy stacks: Protecting return addresses with efficient thread-local storage and runtime re-randomization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–37.

A HOW SEECFI WORKS

Figure 8 shows the structure of SeeCFI when analyzing a single binary file. We used this for debugging and to verify the results of SeeCFI (see Section 5.2). First SeeCFI checks if the binary was compiled using the cross-DSO flag (`multi-module CFI`) based on Algorithm 1. If SeeCFI was invoked using the `-s` flag, the binary is always checked for the pattern of `single-module CFI` using Algorithm 2. Otherwise, SeeCFI only performs this analysis if the multi-module check returns false. In the case of true, the `ShadowCallStack` check, as described in Algortihm 3, is performed immediately. The results are printed after all required checks are finished. It is possible to use the `-d` flag to dump the corresponding assembly code of all single-module CFI and ShadowCallStack occurrences into a separate text file.

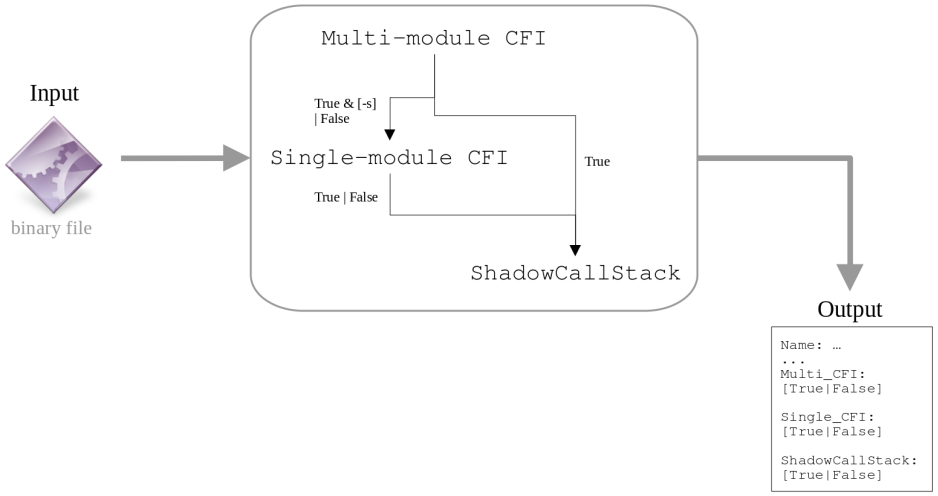

Fig. 8. Analysis structure for a single binary file (used for debugging and verification purposes)

B EXPERIMENT TABLES

Year	Android Versions	Debian Version	Ubuntu Version
2009		5.0 / 5.1	
2011		6.0.0 / 6.0.1	
2012			12.04 / 12.10
2013	4.4	7.0.0 / 7.11.0	13.04 / 13.10
2014			14.04 / 14.10
2015		8.0.0 / 8.11.0	15.04 / 15.10
2016			16.04 / 16.10
2017	7.1.0	9.0.0 / 9.13.0	17.04 / 17.10
2018	(8.1)/9		18.04 / 18.10
2019	10	10.0.0 / 10.13.0	19.04 / 19.10
2020	11		20.04 / 20.10
2021	12 & 12L	11.0.0 / 11.5.0	21.04 / 21.10
2022	13		22.04 / 22.10

Table 1. Release years of Android, Debian, and Ubuntu versions

	OS Version	Vendor	Device	Name
4.4	KRT16M	Android (Google)	Nexus 5 (GSM/LTE)	hammerhead-krt16m
7.1.0	NDE63H	Android (Google)	Pixel XL	marlin-nde63h
9.0.0	PD1A.180720.030	Android (Google)	Pixel 3	blueline-pd1a.180720.030
9.0.0	PD1A.180720.030	Android (Google)	Pixel 3 XL	crosshatch-pd1a.180720.030
9.0.0	PD2a.190115.029	Android (Google)	Pixel 3a	sargo-pd2a.190115.029
9.0.0	PD2a.190115.029	Android (Google)	Pixel 3a XL	bonito-pd2a.190115.029
10.0.0	QD1a.190821.007	Android (Google)	Pixel 4	flame-qd1a.190821.007
10.0.0	QD1a.190821.007	Android (Google)	Pixel 4 XL	coral-qd1a.190821.007
10.0.0	QD4A.200317.027	Android (Google)	Pixel 4a	sunfish-qd4a.200317.027
11.0.0	RD1A.200810.022.A4	Android (Google)	Pixel 4a (5G)	bramble-rd1a.200810.022.a4
11.0.0	RD1A.200810.022.A4	Android (Google)	Pixel 5	redfin-rd1a.200810.022.a4
11.0.0	RD2A.210605.007	Android (Google)	Pixel 5a	barbet-rd2a.210605.007
12.0.0	SD1A.210817.015.A4	Android (Google)	Pixel 6	oriole-sd1a.210817.015.a4
12.0.0	SD1A.210817.015.A4	Android (Google)	Pixel 6 Pro	raven-sd1a.210817.015.a4
12.1.0	SD2A.220601.003	Android (Google)	Pixel 6a	bluejay-sd2a.220601.003
13.0.0	TD1A.220804.009.A2	Android (Google)	Pixel 7	panther-td1a.220804.009.a2
13.0.0	TD1A.220804.009.A2	Android (Google)	Pixel 7 Pro	cheetah-td1a.220804.009.a2
8.1.0	J260GDDS9AVK1_ J260GODM9AVK2	Samsung	Galaxy J2 Core (SM-J260G)	J260GDDS9AVK1_ J260GODM9AVK2_INS
9.0.0	G950USQU8DUD3_ G950UOYN8DUD3	Samsung	Galaxy S8 (SM-G950U)	G950USQU8DUD3_ G950UOYN8DUD3_TMB
10.0.0	G960USQU9FVB2_ G960UOYN9FVB2	Samsung	Galaxy S9 (SM-G960U)	G960USQU9FVB2_ G960UOYN9FVB2_TMB
11.0.0	G973USQU5GUCG_ G973UOYN5GUCG	Samsung	Galaxy S10 (SM-G973U)	G973USQU5GUCG_ G973UOYN5GUCG_TMB
12.0.0	G981USQU2EULH_ G981UOYN2EULH	Samsung	Galaxy S20 (SM-G981U)	G981USQU2EULH_ G981UOYN2EULH_TMB

Table 2. Google and Samsung images (used in Sections 4.2.2 and 4.2.4)

Distribution	OS Version	Image
Debian	5.0 Gnome	debian-live-500-amd64-gnome-desktop.iso
Debian	5.0 Standard	debian-live-500-amd64-standard.img
Debian	5.10 Gnome	debian-live-5010-amd64-gnome-desktop.iso
Debian	5.10 Standard	debian-live-5010-amd64-standard.iso
Debian	6.0 Gnome	debian-live-6.0.0-amd64-gnome-desktop.img
Debian	6.0 Standard	debian-live-6.0.0-amd64-standard.img
Debian	6.0.10 Gnome	debian-live-6.0.10-amd64-gnome-desktop.iso
Debian	6.0.10 Standard	debian-live-6.0.10-amd64-standard.iso
Debian	7.0.0 Gnome	debian-live-7.0.0-amd64-gnome-desktop.iso
Debian	7.0.0 Standard	debian-live-7.0.0-amd64-standard.iso
Debian	7.11 Gnome	debian-live-7.11.0-amd64-gnome-desktop.iso
Debian	7.11 Standard	debian-live-7.11.0-amd64-standard.iso
Debian	8.0.0 Gnome	debian-live-8.0.0-amd64-gnome-desktop.iso
Debian	8.0.0 Standard	debian-live-8.0.0-amd64-standard.iso
Debian	8.11 Gnome	debian-live-8.11.0-amd64-gnome-desktop.iso
Debian	8.11 Standard	debian-live-8.11.0-amd64-standard.iso
Debian	9.0.0 Gnome	debian-live-9.0.0-amd64-gnome.iso
Debian	9.13.0 Gnome	debian-live-9.13.0-amd64-gnome.iso
Debian	10.0.0 Gnome	debian-live-10.0.0-amd64-gnome.iso
Debian	10.0.0 Standard	debian-live-10.0.0-amd64-standard.iso
Debian	10.13.0 Gnome	debian-live-10.13.0-amd64-gnome.iso
Debian	10.13.0 Standard	debian-live-10.13.0-amd64-standard.iso
Debian	11.0.0 Gnome	debian-live-11.0.0-amd64-gnome.iso
Debian	11.0.0 Standard	debian-live-11.0.0-amd64-standard.iso
Debian	11.5.0 Gnome	debian-live-11.5.0-amd64-gnome.iso
Debian	11.5.0 Standard	debian-live-11.5.0-amd64-standard.iso
Ubuntu	12.04.4	ubuntu-12.04.4-desktop-amd64.iso
Ubuntu	12.10	ubuntu-12.10-desktop-amd64.iso
Ubuntu	12.10 mac	ubuntu-12.10-desktop-amd64+mac.iso
Ubuntu	13.04	ubuntu-13.04-desktop-amd64.iso
Ubuntu	13.04 mac	ubuntu-13.04-desktop-amd64+mac.iso
Ubuntu	13.10	ubuntu-13.10-desktop-amd64.iso
Ubuntu	13.10 mac	ubuntu-13.10-desktop-amd64+mac.iso
Ubuntu	14.04.6	ubuntu-14.04.6-desktop-amd64.iso
Ubuntu	14.10	ubuntu-14.10-desktop-amd64.iso
Ubuntu	15.04	ubuntu-15.04-desktop-amd64.iso
Ubuntu	15.10	ubuntu-15.10-desktop-amd64.iso
Ubuntu	16.04.7	ubuntu-16.04.7-desktop-amd64.iso
Ubuntu	16.10	ubuntu-16.10-desktop-amd64.iso
Ubuntu	17.04	ubuntu-17.04-desktop-amd64.iso
Ubuntu	17.10.1	ubuntu-17.10.1-desktop-amd64.iso
Ubuntu	18.04.6	ubuntu-18.04.6-desktop-amd64.iso
Ubuntu	18.04	ubuntu-18.04-desktop-amd64.iso
Ubuntu	18.10	ubuntu-18.10-desktop-amd64.iso
Ubuntu	19.04	ubuntu-19.04-desktop-amd64.iso
Ubuntu	19.10	ubuntu-19.10-desktop-amd64.iso
Ubuntu	20.04.5	ubuntu-20.04.5-desktop-amd64.iso
Ubuntu	20.10	ubuntu-20.10-desktop-amd64.iso
Ubuntu	21.04	ubuntu-21.04-desktop-amd64.iso
Ubuntu	21.10	ubuntu-21.10-desktop-amd64.iso
Ubuntu	22.04.1	ubuntu-22.04.1-desktop-amd64.iso
Ubuntu	22.10	ubuntu-22.10-desktop-amd64.iso

Table 3. Debian and Ubuntu images (used in Section 4.2.2)

	OS Version	Vendor	Device	Name
12.0.0	SP1A.210812.016.C2	Google	Pixel 3	blueline-sp1a.210812.016.c2-factory-fa981d87
12.0.0	SP1A.210812.016.C2	Google	Pixel 3 XL	crosshatch-sp1a.210812.016.c2-factory-27f59137
12.1.0	SP2A.220505.008	Google	Pixel 3a	sargo-sp2a.220505.008-factory-071e368a
12.1.0	SP2A.220505.008	Google	Pixel 3a XL	bonito-sp2a.220505.008-factory-db19d2aa
13.0.0	TP1A.221005.002.B2	Google	Pixel 4	flame-tp1a.221005.002.b2-factory-38e4f49a
13.0.0	TP1A.221005.002.B2	Google	Pixel 4 XL	coral-tp1a.221005.002.b2-factory-db99b1f8
13.0.0	TQ2A.230405.003	Google	Pixel 4a	sunfish-tq2a.230405.003-factory-1cb87bfb
13.0.0	TQ2A.230405.003	Google	Pixel 4a (5G)	bramble-tq2a.230405.003-factory-a854ba24
13.0.0	TQ2A.230405.003	Google	Pixel 5	redfin-tq2a.230405.003-factory-e85b956b
13.0.0	TQ2A.230405.003	Google	Pixel 5a	barbet-tq2a.230405.003-factory-8c94f18d
13.0.0	TQ2A.230405.003.E1	Google	Pixel 6	oriole-tq2a.230405.003.e1-factory-e7997d5a
13.0.0	TQ2A.230405.003.E1	Google	Pixel 6 Pro	raven-tq2a.230405.003.e1-factory-6cadf298
13.0.0	TQ2A.230405.003.E1	Google	Pixel 6a	bluejay-tq2a.230405.003.e1-factory-f21956ab
13.0.0	TQ2A.230405.003.E1	Google	Pixel 7	panther-tq2a.230405.003.e1-factory-ae8e7da5
13.0.0	TQ2A.230405.003.E1	Google	Pixel 7 Pro	cheetah-tq2a.230405.003.e1-factory-1f04869e
12.0.0	SP1A.210812.016.C2	GrapheneOS	Pixel 3	blueline-factory-2023020600
12.0.0	SP1A.210812.016.C2	GrapheneOS	Pixel 3 XL	crosshatch-factory-2023020600
12.1.0	SP2A.220505.008	GrapheneOS	Pixel 3a	sargo-factory-2023020600
12.1.0	SP2A.220505.008	GrapheneOS	Pixel 3a XL	bonito-factory-2023020600
13.0.0	TP1A.221005.002.B2	GrapheneOS	Pixel 4	flame-factory-2023041100
13.0.0	TP1A.221005.002.B2	GrapheneOS	Pixel 4 XL	coral-factory-2023041100
13.0.0	TQ2A.230405.003	GrapheneOS	Pixel 4a	sunfish-factory-2023041100
13.0.0	TQ2A.230405.003	GrapheneOS	Pixel 4a (5G)	bramble-factory-2023041100
13.0.0	TQ2A.230405.003	GrapheneOS	Pixel 5	redfin-factory-2023041100
13.0.0	TQ2A.230405.003	GrapheneOS	Pixel 5a	barbet-factory-2023041100
13.0.0	TQ2A.230405.003.E1	GrapheneOS	Pixel 6	oriole-factory-2023041100
13.0.0	TQ2A.230405.003.E1	GrapheneOS	Pixel 6 Pro	raven-factory-2023041100
13.0.0	TQ2A.230405.003.E1	GrapheneOS	Pixel 6a	bluejay-factory-2023041100
13.0.0	TQ2A.230405.003.E1	GrapheneOS	Pixel 7	panther-factory-2023041100
13.0.0	TQ2A.230405.003.E1	GrapheneOS	Pixel 7 Pro	cheetah-factory-2023041100

Table 4. GrapheneOS and corresponding Google images (used in Section 4.2.3 and 4.2.4)