MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones

Lucas Davi[1], Alexandra Dmitrienko[2], Manuel Egele[3], Thomas Fischer[4],
Thorsten Holz[4], Ralf Hund[4], Stefan Nürnberger[1], Ahmad-Reza Sadeghi[1,2]

[1]CASED/Technische Universität Darmstadt, Germany
{*lucas.davi,stefan.nuernberger,ahmad.sadeghi*}*@trust.cased.de*

[2]Fraunhofer SIT, Darmstadt, Germany
{*alexandra.dmitrienko,ahmad.sadeghi*}*@sit.fraunhofer.de*

[3]University of California, Santa Barbara, USA
maeg@cs.ucsb.edu

[4]Ruhr-Universität Bochum, Germany
{*thomas.fischer,thorsten.holz,ralf.hund*}*@rub.de*

Abstract

Runtime and control-flow attacks (such as code injection or return-oriented programming) constitute one of the most severe threats to software programs. These attacks are prevalent and have been recently applied to smartphone applications as well, of which hundreds of thousands are downloaded by users every day. While a framework for control-flow integrity (CFI) enforcement, an approach to prohibit this kind of attacks, exists for the Intel x86 platform, there is no such a solution for smartphones.

In this paper, we present a novel framework, MoCFI (Mobile CFI), that provides a general countermeasure against control-flow attacks on smartphone platforms by enforcing CFI. We show that CFI on typical smartphone platforms powered by an ARM processor is technically involved due to architectural differences between ARM and Intel x86, as well as the specifics of smartphone OSes. Our framework performs CFI on-the-fly during runtime without requiring the application's source code. For our reference implementation we chose Apple's iOS, because it has been an attractive target for control-flow attacks. Nevertheless, our framework is also applicable to other ARM-based devices such as Google's Android. Our performance evaluation demonstrates that MoCFI is efficient and does not induce notable overhead when applied to popular iOS applications.

1. Introduction

Although control-flow (or runtime) attacks on software are known for about two decades, they are still one of the major threats to software today. Such attacks compromise the control-flow of a vulnerable application during runtime based on diverse techniques (e.g., stack- or heap-based buffer overflows [4, 5], uncontrolled format strings vulnerabilities [23], or integer overflows [9]). Many current systems offer a large attack surface, because they still use software programs implemented in unsafe languages such as C or C++. In particular, modern smartphone platforms like Apple's iPhone and Google's Android have recently become appealing attack targets (e.g., [25, 33, 26, 27, 43]) and increasingly leak sensitive information to remote adversaries (e.g., the SMS database [26]).

A general approach to mitigate control-flow attacks is the enforcement of *control-flow integrity* (CFI) [1]. This technique asserts the basic safety property that the control-flow of a program follows only the legitimate paths determined in advance. If an adversary hijacks the control-flow, CFI enforcement can detect this divagation and prevent the attack. In contrast to a variety of ad-hoc solutions, CFI provides a general solution against control-flow attacks. For instance, to detect conventional return-oriented programming attacks one can check every return instruction the program is executing [12, 21, 16], but recent results show that return-oriented programming without returns is feasible on both x86 and ARM [11]. Moreover, CFI provides stronger protection than recent ASLR (address space layout randomization) mainly due to the fact that existing randomization realizations are often vulnerable to brute-forcing [38] or leak sensitive information about the memory layout [40]. Surprisingly, and to the best of our knowledge, there exist no CFI framework for smartphone platforms.

In this paper, we present the design and implementation of *MoCFI* (Mobile CFI), a CFI enforcement framework for smartphone platforms. Specifically, we focus on the ARM architecture since it is the standard platform for smartphones, and there is currently no smartphone available deploying an x86-based processor [28]. The implementation of CFI on ARM is often more involved than on desktop PCs due to several subtle architectural differences

that highly influence and often significantly complicate a CFI solution: (1) the program counter is a general-purpose register, (2) the processor may switch the instruction set at runtime, (3) there are no dedicated return instructions, and (4) control-flow instructions may load several registers as a side-effect.

Although our solution can be deployed to any ARM based smartphone, we chose Apple's iPhone for our reference implementation because of three challenging issues: First, the iPhone platform is a popular target of control-flow attacks due to its use of the Objective-C programming language. In contrast, Android is not as prone to control-flow attacks because applications are mainly written in the type-safe Java programming language. Second, iOS is closed-source meaning that we can neither change the operating system nor can we access the application's source code. Third, applications are encrypted and signed by default.

Contribution. To the best of our knowledge, *MoCFI* is the first general CFI enforcement framework for smartphone platforms. Solutions like NativeClient (NaCl) for ARM [36] only provide a compiler-generated sandbox and are unsuitable for smartphones. NaCl needs access to source code and currently does not support 16-Bit THUMB code which is typically used in modern smartphone apps. In contrast, our solution operates on binaries and can be transparently enabled for individual applications. *MoCFI* allows us to retrofit CFI onto smartphone applications with commonly unavailable source code. Note that a compile-time solution would be specific to a single compiler, and Apple currently supports two compilers (LLVM and GCC). Hence, compiler-solutions are typically not suitable in practice, since neither the end-user nor the App store maintainer can recompile the application.

To this end, we first implemented a system to recover the *control-flow graph (CFG)* of a given iOS application in binary format. In particular, we extend PiOS [19] (a data-flow analysis framework) to generate the CFG. Based on this information, we perform control-flow validation routines that are used during *runtime* to check if instructions that change the control-flow are valid. Our prototype is based on library injection and in-memory patching of code which is compatible to memory randomization, static code signing, and encryption. Finally, our approach only requires a jailbreak for setting a single environment variable, installing a shared library, and allowing our library to rewrite the application code during load-time.

For performance evaluation, we measured the overhead *MoCFI* introduces as well the average overhead for typical applications and worst-case scenarios. The evaluation shows that our implementation is efficient. Moreover, we proved the effectiveness by constructing a control-flow attack that uses return-oriented programming [37, 10, 20, 24, 30, 11] and techniques similar to GOT (Global Offset Table) dereferencing [45, 22], which our tool can successfully prohibit.

Outline. The remainder of this paper is organized as follows: after briefly recalling the ARM architecture and the iOS smartphone operating system in Section 2, we present the problem of modern control-flow attacks, the original concept of CFI, and technical challenges when applying CFI to smartphones in Section 3. Afterwards, we present the design and implementation of *MoCFI* in Section 4 and 5. In Section 6 we discuss the security of *MoCFI* and current limitations. We present performance measurements in Section 7, summarize related work in Section 8, and conclude the paper in Section 9.

2. Background

In this section, we present a brief overview of the relevant aspects of the ARM processor architecture and the iOS operating system that are closely related to our work.

2.1. ARM Architecture

ARM features a 32 bit processor and sixteen general-purpose registers `r0` to `r15`, where `r13` is used as stack pointer (`sp`) and `r15` as program counter (`pc`). Furthermore, ARM maintains the so-called *current program status register* (`cpsr`) to reflect the current state of the system (e.g., condition flags, interrupt flags, etc.). In contrast to Intel x86, machine instructions are allowed to directly operate on the program counter `pc` (`EIP` on x86).

In general, ARM follows the *Reduced Instruction Set Computer* (RISC) design philosophy, e.g., it features dedicated load and store instructions, enforces aligned memory access, and offers instructions with a fixed length of 32 bits. However, since the introduction of the ARM7TDMI microprocessor, ARM provides a second instruction set called THUMB which usually has 16 bit instructions, and hence, is suitable for embedded systems with limited memory space.

The *ARM architecture procedure call standard* (AAPCS) document specifies the ARM calling convention for function calls [8]. In general, a function can be called by a `BL` (**B**ranch with **L**ink) or `BLX` (**B**ranch with **L**ink and e**X**change) instruction. `BLX` additionally allows indirect calls (i.e., the branch target is stored in a register), and the exchange ("*interworking*") from ARM to THUMB code and vice versa. Both instructions have in common that they store the return address (which is simply the instruction succeeding the `BLX`/`BL`) in the link register `lr` (`r14`). In order to allow nested function calls, the value of `lr` is usually pushed on the stack when the called function is entered.

Function returns are simply accomplished by loading the return address to `pc`. *Any* instruction capable of loading values from the stack or moving `lr` to `pc` can be used as return instruction. In particular, ARM compilers often use "load multiple" instructions as returns meaning that the instruction does not only enforce the return, but also loads several registers, e.g., `POP {R4-R7,PC}` loads `R4` to `R7` and the program counter with new values from the stack.

2.2. Selected Security Features of Apple iOS

Apple iOS is a closed and proprietary operating system designed for mobile Apple devices such as iPhone, iPad, and iPod Touch. A remarkable security feature of iOS is that only binaries and libraries signed by Apple are allowed to execute, which reduces the attack surface for malicious software. Furthermore, Apple only signs applications after inspecting the code. However, Apple provides no information how code inspection is enforced. Further, Apple only has access to the application binary (and not to the actual source code).

Since iOS v2.0, Apple enables the $W \oplus X$ (**W**ritable **xor** e**X**ecutable) security model, which basically marks a memory page either writable or executable. $W \oplus X$ prevents an adversary from launching a code injection attack, e.g., the conventional stack buffer overflow attack [4]. Furthermore, iOS deploys dynamic code signing enforcement (CSE) at runtime [48] to prevent the injection of new (malicious) code. In contrast to systems that only enable $W \oplus X$ (e.g., Windows or Linux), CSE on iOS prevents an application from allocating new memory (e.g., via `mprotect`) marked as executable. On the other hand, CSE at runtime in conjunction with $W \oplus X$ has practical drawbacks because it does not support self-modifying code and code generated by just-in-time (JIT) compilers. Therefore, iOS provides the so-called *dynamic-codesigning* entitlement that allows applications to generate code at runtime. At the time of writing, only the Mobile Safari Browser and full-screen web applications are granted the *dynamic-codesigning* entitlement. However, a very recent attack demonstrates that the current CSE implementation is vulnerable allowing an adversary to apply the *dynamic-codesigning* to arbitrary market applications [35]. Moreover, neither CSE at runtime nor $W \oplus X$ can prevent return-oriented programming attacks that only leverage existing and signed code pieces.

To detect stack-based buffer overflow attacks, iOS deploys the Stack-Smashing Protector (SSP). Basically, SSP uses *canaries*, i.e., guard values that are placed between local variables and control-flow information to detect simple stack smashing attacks. Moreover, SSP features bounds-checking for selected critical functions (like `memcpy` and `strcpy`) to ensure that their arguments will not lead to stack overflows. However, bounds-checking is only performed for a limited set of functions and SSP cannot detect heap overflows or any other control-flow attack beyond stack smashing.

A very recent feature of iOS (since iOS v4.3) is *address space layout randomization* (ASLR). Basically, ASLR randomizes the base addresses of libraries and dynamic areas (such as stack and heap) thereby preventing an adversary from guessing the location of injected code (or useful library sequences). However, it has been shown that existing randomization realizations are vulnerable to various attacks [38, 40, 13]. In addition, we present an own developed iOS exploit (see Appendix B) that successfully circumvents ASLR on iOS.

3. Problem Description

In the following, we discuss the problem of modern control-flow (runtime) attacks, present the basic idea of control-flow integrity (CFI), and finally elaborate on the technical obstacles to overcome when designing CFI enforcement for smartphone platforms.

3.1. Control-Flow Attacks

Figure 1 depicts a sample *control-flow graph* (CFG) of an application. Basically, the CFG represents valid execution paths the program may follow while it is executing. It consists of *basic blocks* (BBLs), instruction sequences with a single entry, and exit instruction (e.g., return, call, or jump), where the exit instruction enables the transition from one BBL to another. Any attempt of the adversary to subvert the valid execution path can be represented as a deviation from the CFG, which results in a so-called *control-flow* or *runtime* attack.

In particular, Figure 1 illustrates two typical control-flow attacks at BBL3: (i) a *code injection attack* (transition 2a), and (ii) a *code reuse attack* (transition 2b). Both attacks have in common that the control-flow is not transferred to BBL 5, but instead to a piece of code not originally covered by the CFG. A conventional control-flow attack is based on the injection of malicious code into the program's memory space. For instance, the adversary may overflow a buffer on the stack by first injecting his own malicious code and then overwriting a function's return address with the start address of the injected code [4]. However, modern operating systems (such as iOS) enforce the $W \oplus X$ security model that prevents an adversary from executing injected code. On the other hand, code-reuse attacks such as return-into-libc [34, 39] and modern return-oriented programming (ROP) [37, 10, 20, 24, 30, 11] bypass $W \oplus X$ by redirecting execution to code already residing in the program's memory space. In particular, ROP allows an adversary to induce

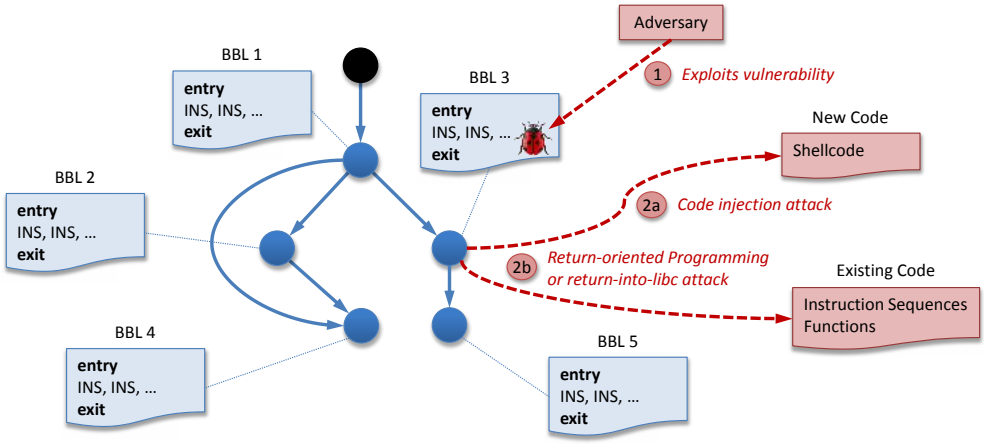

Figure 1. Schematic overview of control-flow attacks

arbitrary program behavior by only combining small code pieces from different parts of linked libraries.

Recent news underline that control-flow attacks are a severe problem on smartphones. In particular, control-flow attacks can be utilized to steal the user's SMS database [26], to open a remote reverse shell [27], or to launch a jailbreak [13]. Unfortunately, there is no general countermeasure to defeat such attacks on smartphones.

3.2. Control-Flow Integrity (CFI) on x86

A general approach to defeat control-flow attacks is the enforcement of control-flow integrity (CFI) [1]. Basically, CFI guarantees that a program *always* follows a valid execution path in the CFG, where the CFG is created ahead of time by means of static analysis. This is achieved by checking branch instructions of the BBLs: such instructions are overwritten with a new block of instructions that validates if the branch instruction targets a valid execution path in the CFG. For instance, in Figure 1, CFI would check if the exit instruction of BBL3 targets BBL5. In particular, the CFI prototype implementation presented by Abadi et al. [1] inserts unique labels just above each entry instruction. Hence, the CFI validation routine only has to check whether the branch address targets an instruction that is preceded by a valid label. However, the adoption and adaption of CFI (as presented in [1]) to smartphone platforms involves several difficulties and challenges.

3.3. Technical Challenges on Smartphone Platforms

The technical challenges are due to the architectural differences between ARM (RISC design) and Intel x86 (CISC design), and because of the specifics of smartphone operating systems. These highly influence and often complicate a CFI solution as we argue in the following.

No Dedicated Return Instruction. As mentionend in Section 2.1, ARM does not provide dedicated return instructions. Instead, any branch instruction can be used as a return. Moreover, returns may have side-effects, meaning that the return does not only enforce the return to the caller, but also loads several registers within a single instruction. Hence, in contrast to Intel x86, a CFI solution for ARM has to handle all different kinds of returns, and has to ensure that all side effects of the return are properly handled.

Multiple Instruction Sets. CFI on ARM is further complicated by the presence of two instruction sets (ARM and THUMB), which can even be interleaved. Hence, it is necessary to distinguish between both cases during the analysis and enforcement phase, and to ensure the correct switching between the two instruction sets at runtime.

Direct Access to the Program Counter. Another difference is that the ARM program counter `pc` is a general-purpose register which can be directly accessed by a number of instructions, e.g., arithmetic instructions are allowed to load the result of an arithmetic operation directly into `pc`. Furthermore, a load instruction may use the current value of `pc` as base address to load a pointer into another register. This complicates a CFI solution on ARM, since we have to consider and correctly handle all possible control-flow changes, and also preserve the accurate execution of load instruction that use `pc` as base register.

Application Signing and Encryption. Smartphone operating systems typically feature application encryption and signing. Since the traditional CFI approach [1] performs changes on the stored binary, the signature of the application cannot be verified anymore.

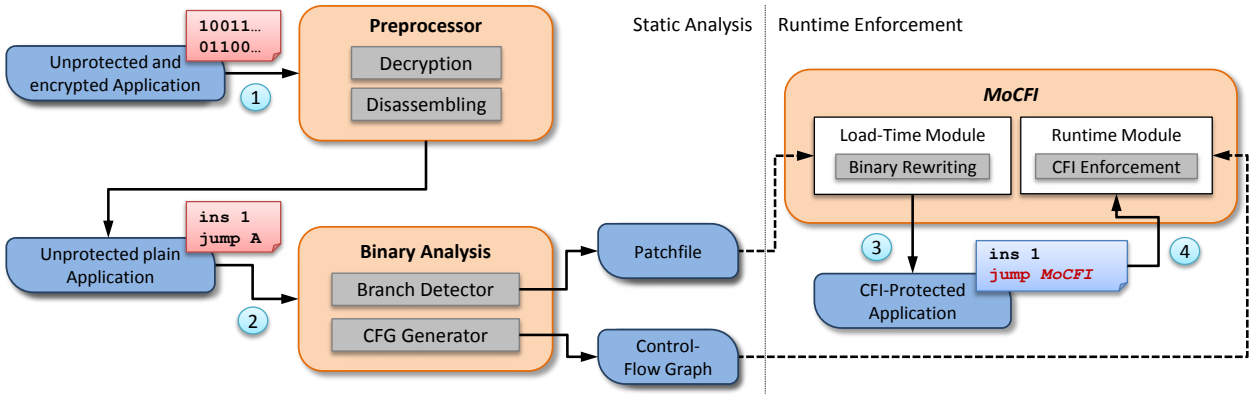

Figure 2. Control-flow integrity for smartphone applications

Closed-Source OS. Several smartphone OSes (such as iOS) are closed-source. Hence, we cannot change the actual OS to deploy CFI on smartphones. Moreover, end-users and even App Store maintainers (e.g., Apple's App Store) have no access to the application source code. Hence, a compiler-based CFI solution is not practicable from the end-user's perspective.

4. Design of our CFI Framework

Our general architecture is shown in Figure 2. From a high-level point of view, our system is separated in two different phases: static analysis and runtime enforcement. The *static* tools perform the initial analysis of the compiled application binary file. In the preprocessing phase, we first decrypt and disassemble the binary (step 1). Afterwards, we thoroughly analyze the application binary and its structure (step 2): In particular, we generate the control-flow graph (CFG) of the application and employ a branch detector to identify all branches contained in the binary and extract all information that is necessary to enforce CFI at runtime. Note that these steps have to be performed only *once* after compilation and can be integrated as an additional step in the deployment phase of a typical smartphone application. Finally, we monitor the application at *runtime* by applying our *MoCFI* shared library that rewrites the binary at load-time (step 3) and enforces control-flow restrictions while the application executes (step 4).

Although the depicted design applies in general to all CFI solutions, our design requires a number of changes, mainly due to (i) the architectural differences between ARM and Intel x86, (ii) the missing binary rewriter and automatic graph generation for ARM, and (iii) the specifics of smartphone operating systems. In the following we describe each involved system component and our approach in more detail.

Preprocessor. The first step of our static analysis phase is performed within the preprocessor component, which has mainly two tasks: (1) decrypting, and (2) disassembling the target application binary. In particular, we faced the challenge that smartphone applications are often encrypted by default (e.g., iOS applications). We thus obtain the unencrypted code of a binary through *process dumping* [19]. For disassembling the application binary we deploy standard disassembler tools that support the ARM architecture.

Binary Analysis. The original CFI work for Intel x86 [1] employs the binary instrumentation framework Vulcan [18] to automatically derive the CFG and to statically rewrite an application binary. However, such a framework does not exist for ARM. Hence, we developed own techniques to accurately generate the CFG. After our preprocessor decrypted and disassembled the application binary, we identify all relevant branches contained in the binary. By relevant branches, we refer to branch instructions that an adversary may exploit for a control-flow attack. These mainly comprise indirect branches, such as indirect jumps and calls, and function returns. Moreover, we include direct function calls to correctly validate function returns, i.e., to be able to check if a function return targets the original caller. We do not instrument direct jump instructions for obvious reasons: the target address for these are fixed (hard-coded), and hence cannot be manipulated by an adversary. Finally, we store meta information for each indirect branch and function call (e.g., instruction address, length, type, etc.) in a separate patchfile.

Based on the result of the branch detector, we generate the CFG by static tools that we developed ourselves. In particular, our static tools calculate possible target addresses for each indirect branch. Finally, a binary representation of the CFG is stored in a separate file (denoted as Control-Flow Graph), which is linked to the smartphone application at runtime.

***MoCFI* Load-Time Module: Binary Rewriting.** The binary rewriting engine is responsible for binding additional code to the binary that checks if the application follows a valid path of the CFG. Typically, one replaces all branch instructions in the binary with a number of new instructions that enforce the control-flow checks [1]. However, replacing one instruction with multiple instructions requires memory adjustments, because all instructions behind the new instructions are moved downwards. The Intel x86 approach uses the Vulcan binary instrumentation framework [18] which automatically accomplishes this task. However, memory adjustment without a full binary rewriting framework requires high implementation efforts.

Due to the limited possibilities to change smartphone binaries (due to code signing) and the missing full binary rewriter, we opted for the following rewriting approach (which has been originally proposed by Winwood et al. [44]). At *load-time* we replace all relevant branches (based on the extracted rewriting) with a single instruction: the so-called *dispatcher instruction*. The dispatcher instruction redirects the control-flow to a code section where the CFI checks reside, namely to the runtime module of our *MoCFI* shared library.

This approach also raises several problems: First, accurate branch instructions have to be implemented that are able to jump to the correct CFI check. Second, the CFI checks require information from where the dispatch originated. As we will demonstrate in the rest of the paper, our solution efficiently tackles the above mentioned problems.

***MoCFI* Runtime Module: Control-Flow Integrity Enforcement.** The key insight of CFI is the realization of control-flow validation routines. These routines have to validate the target of every branch to prevent the application from targeting a BBL beyond the scope of the CFG and the current execution path. Obviously, each branch target requires a different type of validation. While the target address of an indirect jump or call can be validated against a list of valid targets, the validation of function returns requires special handling because return addresses are dynamic and cannot be predicted ahead of time. To address this issue, *MoCFI* reuses the concept of shadow stacks that hold valid copies of return addresses [12], while the return addresses are pushed onto the shadow stacks when a function call occurs.

5. Implementation Details

Our prototype implementation targets iOS 4.3.1, and we successfully applied to 4.3.2 as well. We developed the static analysis tools (842 lines of code) with the IDC scripting language featured by the well-known disassembler IDA Pro 6.0. Moreover, we used Xcode 4 to develop the *MoCFI* library (1,430 lines of code). Our prototype implementation currently protects the application's main code, but no dynamic libraries that are loaded into the process. Hence, an adversary may launch a control-flow attack by exploiting a shared library. We leave support for shared libraries open to future work. However, it is straightforward to extend *MoCFI* accordingly, there are no new conceptional obstacles to overcome. We now describe how we generate the CFG and the patchfile of an iOS binary, and in particular present implementation details of our *MoCFI* library.

5.1. Static Analysis

Since iOS restricts access to source code, we apply our static analysis techniques directly on iOS binaries to generate the CFG and to identify all branches in the binary. We need the former one to validate if a branch follows a valid execution path, while the latter one is used to guarantee accurate binary rewriting. To perform this task, we use the IDA Pro v6.0 Disassembler that enables us to accurately disassemble ARM and THUMB code. Specifically, we implemented IDA scripts to automate the analysis and extract the necessary information from a given binary.

Patchfile Generation. As shown in Figure 2 in Section 4, step 2 involves the generation of rewriting information for each individual binary. This information is required by the load-time module of *MoCFI* to replace each branch instruction with a new instruction that redirects execution to the accurate CFI validation routine. In order to identify all relevant branch instructions, we evaluate each instruction belonging to the text segment and check if the instruction is relevant in the context of CFI. Afterwards, we perform a fine-grained instruction analysis and store the derived meta information (e.g. instruction address, mode, length, type, etc.) in the patchfile. By bundling the patchfile with the application, we can protect its integrity, as all application bundle contents are code-signed.

Generation of the Control-Flow Graph. In order to generate the CFG, we utilize IDA Pro to divide the binary into basic blocks (BBLs, see Section 3) and gather all assembly instructions that divert the control-flow. These instructions can be divided in two categories: (1) instructions that contain their possible control-flow destination as an immediate value, (2) instructions that continue control-flow based on the value of a register. While the first type is trivial in a $W \oplus X$ environment (i.e., the destination cannot be changed), the latter case can only be checked during runtime. Hence, as an optimization step and as argued in Section 4, we remove type (1) branches from the CFG. Type (2) branches are more challenging as the value they depend on needs to be calculated during static analysis. If this is not

possible, heuristics have to be applied at runtime to narrow the possible control-flow destinations. Therefore, it is impossible to construct the CFG in all cases. Note that this is a general shortcoming of all CFG generation methods and not specific to our approach.

Indirect Branches and Heuristics.

To calculate the target address of an indirect branch (e.g., `LDR pc,[r2,r3,LSL #2]`), the registers have to be tracked backwards and all of their possible values must be calculated. In the above example, the target (`pc`) is calculated as $\texttt{pc} \leftarrow \texttt{r2} + \texttt{r3} \cdot 4$. If `r2` and `r3` can be tracked, the correct value can be extracted.

```
0x1000: MOV     r2, 0x2000
0x1004: ADD.W   r2, r2, r3, LSL#2
0x1008: MOV     pc, r2

0x2000: B.W     0x3000
0x2004: B.W     0x3100
0x2008: B.W     0x3200
```

Listing 1. Indirect Jump with jump table

Listing 1 is a common compiler-generated pattern to optimize `switch`-statements. Depending on `r3`, the possible control-flow targets (`pc`) are `0x3000, 0x3100`, and `0x3200` and our analysis can recover these possibilities.

In case an indirect branch cannot be resolved (e.g. hand-written assembler code), it is possible to apply a heuristic in order to constrain the control-flow. A general constraint on ARM is that the target address must be a multiple of the instruction length. For indirect calls this constraint can be narrowed down to the beginning of functions. Even though it is still possible to call arbitrary functions, the target control-flow cannot land inside a function body.

Indirect jumps (`BX`) can be constrained to only take place inside one function body, i.e., not crossing function boundaries. Despite there is no technical need for this restriction, the `C`, `C++` and `Objective-C` languages restrict control-flow to the scope of one function, with the exception of function calls. Hence, we can assume that the scope of indirect jumps is limited by the boundary of a function body.

Objective-C Peculiarities. Traditional CFG generation techniques also need to be extended for iOS applications due to a peculiarity of Objective-C. Internally, any method call of an Objective-C object is resolved to a call to the generic message handling function `objc_msgSend`. The name of the actual method (called *selector*) is given as a parameter. Consequently, we must track these parameters in the CFG generation phase and include them in the CFG. Otherwise, an adversary might mount an attack by modifying the method parameters of `objc_msgSend`, thus diverting the control-flow to an invalid method. We built upon PiOS [19] and former reverse-engineering work on iOS [17, 32] to generate CFG information for `objc_msgSend` calls.

5.2. Control-Flow Integrity Checking with *MoCFI*

Most UNIX-based operating systems support the injection of libraries by providing the environment variable `LD_PRELOAD` that is checked by the OS loader upon initialization of each new process. The loader ensures that the library is loaded before any other dependency of the actual program binary. iOS provides an analogous method through the `DYLD_INSERT_ LIBRARIES` environment variable [6]. Setting this variable to point to *MoCFI* enables us to transparently instrument arbitrary applications. To force loading *MoCFI* into every application started though the touchscreen, we only need to set this environment variable for the *SpringBoard* process.

We stress that *MoCFI* is initialized *before* any other dependency of the program and *after* the signature of the application was verified (i.e., the program binary itself is unaltered). Subsequently, *MoCFI* implements the CFI enforcement by rewriting the code of the application in memory. We call this part of *MoCFI* the *load-time module* in contrast to the *runtime* module, which holds the actual validation checks. The load-time module makes sure that all relevant control-flow instructions are diverted to a corresponding validation routines in *MoCFI*. If validation succeeds, execution continues at the desired branch location.

5.2.1. Load-Time Module: Binary Rewriting

Upon initialization of *MoCFI*, the load-time module first locates the correct patchfile. Afterwards, it rewrites the binary according to the information stored in the patchfile. Since Apple iOS enforces $W \oplus X$, code cannot be writable in memory. Therefore, the code pages must be set to writable (but not executable) first. This is usually accomplished using the POSIX-compliant `mprotect` system call. Our experiments revealed that iOS does not allow the code pages to be changed at all (`mprotect` returns a permission denied error). However, this problem can be overcome by re-mapping the corresponding memory areas with `mmap` [7] first. When all relevant instructions of a page are patched, the page permissions are set back to executable but not writable. Note that the presence of `mmap` does not give an adversary the opportunity to subvert *MoCFI* by overwriting code. In order to do so, he would have to mount an attack beforehand that would inherently violate CFI at some point, which in turn would be detected by *MoCFI*.

Trampolines. Our binary rewriting engine overwrites the relevant control-flow instructions with the so-called *dis-*

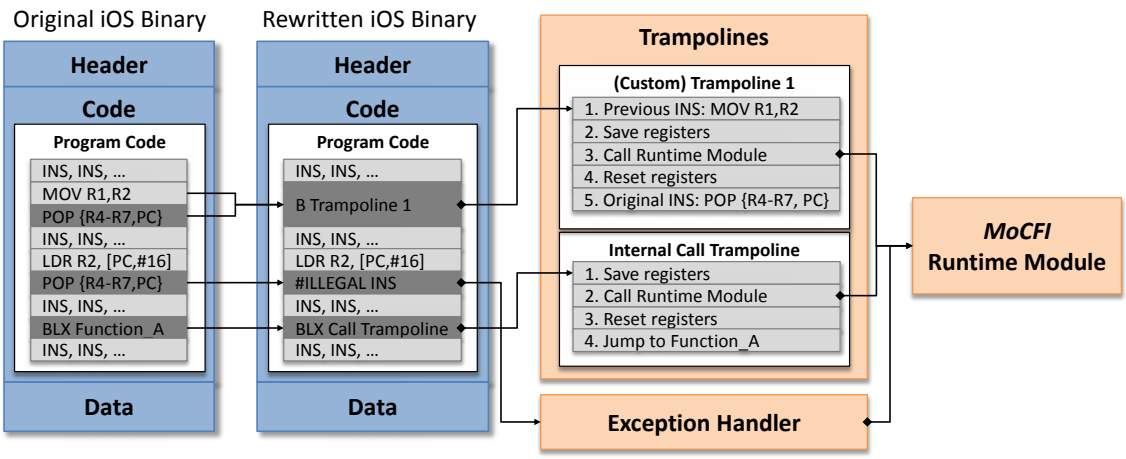

Figure 3. Trampoline approach

patcher instructions. The dispatcher redirects the program flow to a short piece of assembler code, namely to the *trampoline*, which in turn transfers the execution to our *MoCFI* library (see Figure 3). Hence, the trampolines are used as bridges between the application we aim to protect and our *MoCFI* library.

Specifically, we allocate dedicated trampolines for each indirect branch (i.e., indirect jumps / calls and returns), one generic trampoline for direct internal function calls (i.e., calls within the same code segment), and one generic trampoline for external function calls. Two example trampolines are shown in Figure 3: the first one (Trampoline 1) is used for a return instruction, while the second one (Internal Call Trampoline) handles a (direct) internal function call. In general, each trampoline saves the current execution state, invokes the appropriate *MoCFI* validation routine, resets the execution state, and issues the original branch. Due to the last step, we ensure that all registers are loaded correctly, even if the branch loads several registers as a side-effect, e.g., the replaced return `POP {R4-R7,PC}` is copied by our load-time module at the end of Trampoline 1. Hence, we ensure that `r4` to `r7` are correctly loaded with values from the stack before the return address is loaded to `pc`.

Note that, depending on the replaced branch instruction, we allocate a THUMB or ARM trampoline to ensure the correct interworking between the two instruction sets. In the following, we present the different kinds of dispatcher instructions our solution utilizes. The specific implementation of the different ARM/THUMB trampolines is described in Appendix A.

ARM (32 bit) dispatcher instruction. Since each ARM instruction is 32 bit long, we can use the default branch (`B`) instruction as a generic dispatcher instruction, which yields a possible target range from -2^{24} to $2^{24}-1$ bytes (32MB). As a consequence, the dispatcher target cannot lie within arbitrary memory address ranges. We address this problem by jumping to the mentioned *trampolines* rather than directly to the validation routine. The trampolines are small in size and are allocated near the code section of the application using `mmap`. Note that, theoretically, an application's memory image could be too large to find a free memory page for the trampolines. However, the application image would have to be larger than 16MB in order to break this approach. In practice, this is very unlikely. Even if this would be the case, one could search for memory regions within the code section that are unused (e.g., due to alignment) and use them as trampolines.

Note that for direct function calls, we use the `BLX` instruction instead, which provides the same target range as the `B` instruction. Using `BLX` ensures that the return address of the call is moved into the `lr` (link) register[1]. Further, *MoCFI* can easily lookup the original call target (stored in our patchfile) by inspecting the link register.

THUMB (16 bit and 32 bit) dispatcher instructions. For THUMB instructions, the situation is more complicated. While immediate branches are typically 32 bits in size (i.e., one can use the same approach as discussed previously), many control-flow instructions exist that are only 16 bits long. As a consequence, the reduced range for a branch target (only -2^{11} to $2^{11}-1$, i.e., 1KB) demands for a different solution. *MoCFI* addresses this issue by replacing a 16 Bit indirect branch with a 32 Bit dispatcher instruction. However, this has the effect that we overwrite 2 Thumb in-

[1]Note that for indirect calls *MoCFI* uses `B` as dispatcher instruction and correctly sets `lr` within the validation routine, because the value of `lr` for indirect calls is dependent on whether the call is an external or internal call. For external calls we let `lr` point to a specialized code piece of *MoCFI* to recognize when external (indirect) library calls returned (see also Appendix A).

structions: the original branch (`POP {R4-R7,PC}`) and the instruction preceding the branch (`MOV R1,R2`). To preserve the program's semantics, we execute the latter one at the beginning of our trampolines (step 1 in Trampoline 1).

Dispatching through exception handling. However, replacing 2 THUMB instructions is not possible if the instruction preceding the branch references the program counter or is itself a branch. For instance, `LDR R2,[PC,#16]` in Figure 3 uses the current value of `pc` to load a pointer. Note that such instructions are not allowed on Intel x86. In such scenarios, we use an entirely different approach: upon initialization, we register an iOS exception handler for illegal instructions. The dispatcher instruction is then simply an arbitrary illegal instruction that will trigger our exception handler. Since this technique induces additional performance overhead, we only use it for exceptional cases. To further reduce the use of the exception handler, one could calculate the address from which `pc` is loaded in the static analysis phase and replace the relevant load instruction with a new memory load instruction which could be placed at the beginning of the trampoline.

Note that our exception handler forwards all exceptions not caused by *MoCFI*. Furthermore, by monitoring the exception handling API, we can ensure the position of our handler in the exception chain.

5.2.2. Runtime Module: CFI Enforcement

An abstract view of the runtime module is shown in Figure 4: it mainly consists of dedicated validation routines for each branch type, where each type is represented by a rectangle on the left side of Figure 4. The validation routines have to validate the target of every branch to prevent the application from targeting a BBL beyond the scope of the CFG and the current execution path. Obviously, each branch target requires a different type of validation, as we will describe in the following.

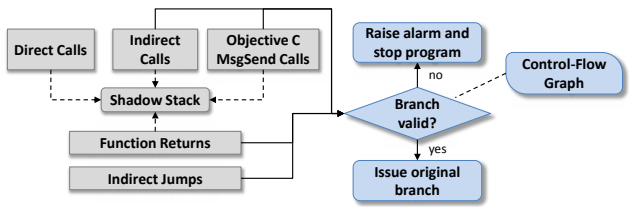

Figure 4. Overview of the runtime module

Function Calls and Returns. To prevent return-oriented attacks, we monitor all function calls and returns, and apply the *shadow stack paradigm* [12]: whenever the program invokes a subroutine (through a direct, indirect, or dispatcher call), we copy the return address on a dedicated shadow stack. Upon function return, we compare the return address the program wants to use to the address stored on our shadow stack. Since function calls (through `BL` or `BLX`) automatically store the return address in `lr`, we simply need to push `lr` onto the shadow stack. Further, we maintain a separate shadow stack for each execution thread to support multi-threaded programs. Upon function return, we determine the return address the program aims to use and retrieve the required stack pointer offset from the patchfile.

As mentioned in Section 3.3, return instructions can be implemented in many different ways on ARM, and often involve the loading of several general-purpose registers. We ensure that all side-effects are correctly handled by issuing the original return at the end of the trampoline (as described in Section 5.2.1).

Indirect Jumps and Calls. The possible jump targets for indirect jumps and calls have either been calculated during static analysis (see Section 5.1) or remain completely unknown. In the first case, the pre-calculated values have to be compared to the outcome of the instruction that *MoCFI* intercepted. In the most complex and versatile form, the instruction is of the form `LDR pc,[rX,rY,LSL#z]` which loads `pc` according to the given register values: $pc \leftarrow r_x + r_y \cdot 2^z$. Consequently, *MoCFI* checks the current value of the registers according to the above equation to match one of the saved, valid jump targets. The information which registers are used by the indirect jump is saved in our patchfile. However, for simpler indirect jumps such as `MOV pc,rX` and indirect calls (`BLX rX`) we simply have to check the content of `rX`. In case the required information cannot be calculated in advance during the static analysis phase, we use heuristics (see Section 5.1) to ensure that the jump targets reside inside the scope of the current function, or for indirect calls, target a valid function prologue.

Objective C MsgSend Calls. Dispatcher calls via the `objc_msgSend` function work like indirect calls. However, instead of a register, they use the function's name (*selector*) and *class instance* to refer to a function's implementation address (see Section 5.1). We trust the implementation of `objc_msgSend` and do not check whether it resolved the correct address. We rather check the supplied parameters *selector* and *class instance*. Both is necessary as the emitted code by GCC usually de-references a register and passes its de-referenced value to `objc_msgSend`. However, the referenced memory is writable and could be overwritten by an adversary. The selector is a simple zero-terminated C-string. Checking whether the selector is still correct can be verified by comparing the original string extracted during static analysis to the current string *selector* points to.

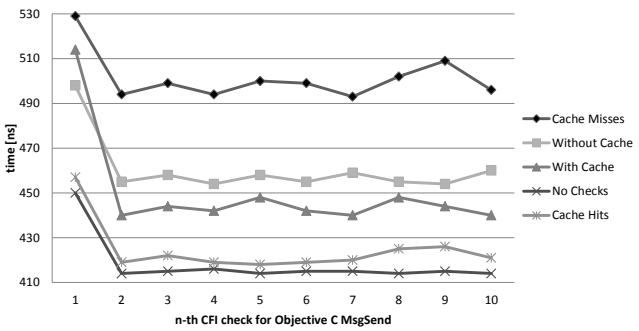

Figure 5. Objective C string comparison optimization using a Cache

We enhanced the performance by introducing a string cache (see Figure 5). This is possible as the strings themselves are write-protected. This enables us to cache the result of the comparison based on a pointer to that string.

For the class instance checking, the name of the class has been extracted by the static analysis. This name is of the form `_OBJC_CLASS_$_NSObject`, where `NSObject` is the class name. The *Runtime Module* obtains the load address of every symbol by using `dlsym()`. The class instance parameter supplied to `objc_msgSend` must then either directly point to that symbol address (static class) or is an instance of that class. For class instances, the first word (32 bit pointer) points to the symbol address of the corresponding class.

6. Discussion and Security Considerations

Our solution adheres to the goal of detecting deviations from the control-flow at runtime from the known-good control-flow. Since iOS enforces $W \oplus X$, a memory page cannot be writable and executable at the same time. Hence, it suffices to check branch targets that can be changed during runtime as they depend on the value of a variable. These include indirect branches and returns from function calls, as they use an address popped from a potentially tampered stack. As we check each such instruction, an adversary cannot subvert the control-flow without *MoCFI* noticing it. However, not all valid, known-good targets can be calculated in advance during the static analysis phase. If this is not the case, we need to apply heuristics to hinder (or at least minimize) their impact. Fortunately, for the majority of tested applications, the indirect branches are used in conjunction with jump tables (see Section 5.1), and can be resolved during the static analysis phase. Moreover, our static tools could be extended by enhanced backtracking techniques to limit the set of possible branch targets. However, the design of sophisticated static tools is not within the scope of this paper, instead our focus is a framework that provides the foundation for system-wide and efficient CFI enforcement on smartphone platforms with an underlying ARM processor.

Since *MoCFI* performs binary rewriting after the iOS loader has verified the application signature, our scheme is compatible to application signing. On the other hand, our load-time module is not directly compatible to the iOS CSE (code signing enforcement) runtime model (see Section 2.2). CSE prohibits any code generation at runtime on non-jailbroken devices, except if an application has been granted the *dynamic-signing* entitlement. To tackle this issue, one could assign the *dynamic-signing* entitlement to applications that should be executed under the protection of *MoCFI*. On the one hand, this is a reasonable approach, since the general security goal of CFI is to protect benign applications rather than malicious ones. Further, the *dynamic-signing* entitlement will not give an adversary the opportunity to circumvent *MoCFI* by overwriting existing control-flow checks in benign applications. In order to do so, he would have to mount a control-flow attack beforehand that would be detected by *MoCFI*. On the other hand, when *dynamic-signing* is in place, benign applications may unintentionally download new (potentially) malicious code, or malicious applications may be accidentally granted the *dynamic-signing* entitlement (since they should run under protection of *MoCFI*) and afterwards perform malicious actions. To address these problems, one could constrain binary rewriting to the load-time phase of an application, so that the *dynamic-signing* entitlement is not needed while the application is executing. Further, new sandbox policies can be specified that only allow the *MoCFI* library to issue the `mmap` call to replace existing code, e.g., the internal page flags of the affected memory page are not changed, or their values are correctly reset after *MoCFI* completed the binary rewriting process.

Finally, special care must be taken that an adversary cannot tamper with the *MoCFI* library and thus bypass *MoCFI*. Since our library is small in size, the probability for exploitable vulnerabilities is very low. Given the small code base, we could also apply code verification tools.

Limitations. Similar to CFI for Intel x86, our current implementation does not detect attacks exploiting exception handlers: an adversary can overwrite pointers to an exception handler and then deliberately cause an exception (e.g., by corrupting a pointer before it is de-referenced). This is possible because GCC pushes these pointers on the stack on demand. We stress that this is rather a shortcoming of the iOS operating system — similar problems have already been solved on other platforms, such as on Windows [31]. Therefore, we encourage Apple to port these techniques to iOS.

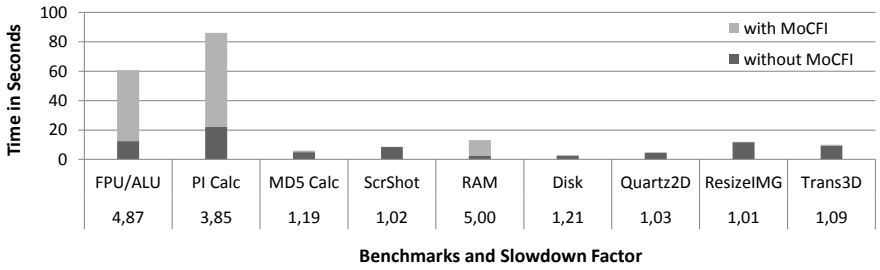

Figure 6. Gensystek Lite Benchmarks

As already mentioned in Section 5, *MoCFI* does currently not protect shared libraries, which an adversary may exploit to launch a control-flow attack. However, extending *MoCFI* accordingly is straightforward, because we already overcame the conceptional obstacles. Note that Apple also explicitly discourages developers to employ shared libraries. Hence, all loaded libraries are typically from the operating system and Objective C frameworks.

Consequently, we currently disable the return check if an external library calls a function that resides in the main application. Therefore, *MoCFI* registers when execution is redirected to a library and disables the return address check for functions that are directly invoked by the shared library. However, note that this can be easily fixed by either applying our trampoline approach to function prologues (i.e., pushing the return address on the shadow stack at function entry) or by applying *MoCFI* to shared libraries.

7. Evaluation

In order to evaluate the performance of *MoCFI*, we applied it to an iOS benchmark tool (called Gensystek Lite[2]), applied it to a full-recursive own developed quicksort algorithm, and performed micro benchmarks to measure the overhead for each type of branch. As we described in Section 6, we apply *MoCFI* to the main application code, and not to the libraries. However, the benchmark tools we apply perform most part of the computation within the application.

Figure 6 shows the results for the Gensystek Lite application, where the slowdown factor for each individual benchmark is shown at bottom of the x-axis. Remarkably, the FPU/ALU, PI calculation, and the RAM (memory read-/write) benchmarks add the highest overhead. However, their slowdown is still reasonable (3.85x and 5x, respectively) considering that FPU/ALU and PI calculations are computationally intensive tasks. The overhead (slowdown greater than factor 1) for the remaining benchmarks is very low and ranges between 1% to 21%.

[2] `http://www.ooparts-universe.com/apps/app_gensystek.html`

n	Without *MoCFI*	With *MoCFI*
100	0.047 ms	0.432 ms
1000	0.473 ms	6.186 ms
10000	6.725 ms	81.163 ms

Table 1. Quicksort

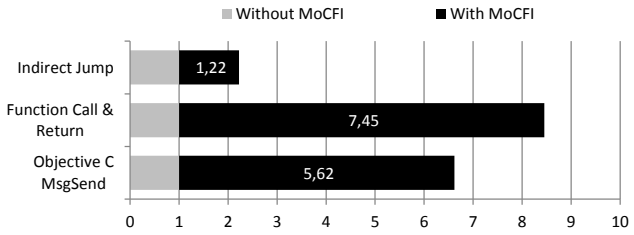

Figure 7. Overhead for each branch type

In order to approximate an upper boundary for performance penalties, we evaluated *MoCFI* by running a quicksort algorithm. Our implemented algorithm makes use of recursion and continuously calls a compare function which consists of only 4 instructions and one *return*. Therefore, *MoCFI* frequently performs a control-flow check in this worst-case scenario. Nevertheless it performs quite well and needs 81ms for $n = 10,000$ (see Table 1).

In order to evaluate the overhead of an instruction that has been replaced by a CFI check, we measured the execution time of three typical instructions and their replacement by *MoCFI* (see Figure 7). The calculation of the overhead per replaced instruction is depicted in Figure 8. For the exemplary case of *Function Calls and Returns*, the ac-

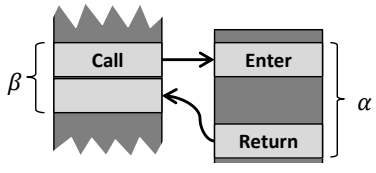

Figure 8. Overhead calculation

n	50	100	500
ψ **(Indirect Jump)**	2.4%	1.2%	0.2%
ψ **(Func. C. & R.)**	14.6%	7.4%	1.5%
ψ **(ObjC MsgSend)**	11.0%	5.6%	1.1%

Table 2. Total overhead ψ

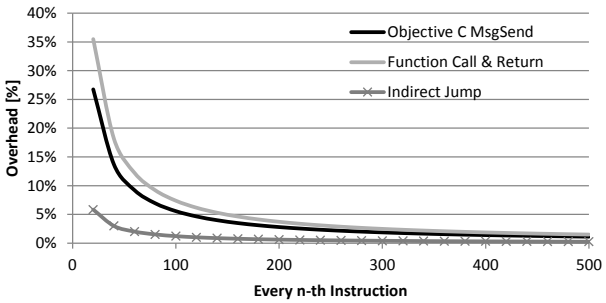

Figure 9. Average overhead

tual function bodies (α) are subtracted from the time measurement between *call* and *return* (β). The actual execution time is therefore $\beta - \alpha$. When running with *MoCFI*, the measurements (α' and β', respectively) are set in relation to the measurement without *MoCFI*. Hence, the instruction slowdown factor φ for one function replaced by *MoCFI* is $\varphi = \frac{\beta' - \alpha'}{\beta - \alpha}$. For our tests, all the measurements have been conducted $10,000$ times and averaged. However, in a typical program, instructions that have to be checked by *MoCFI* are surrounded by other instructions. For n instructions in between, *MoCFI* only has to be called every $(n+1)$-th instruction. The total slowdown ψ is therefore $\psi = \frac{n+\varphi}{n+1}$. The overhead $(\psi - 1)$ as a function of n (instructions between CFI checks) is plotted in Figure 9. An exemplary table with exact values resulting from of $n = 50, n = 100$ and $n = 500$ instructions between CFI checks is depicted in Table 2.

Moreover, we applied *MoCFI* to several popular iOS applications, among others Facebook, Minesweeper, TexasHoldem, and Gowalla. Our experiments showed that *MoCFI* does not induce any notable overhead while the applications execute. Further, *MoCFI* induces an acceptable overhead at load-time: e.g., for the Facebook application (code size 2.3MB; 33,647 calls; 5,988 returns; 20 indirect jumps) and TexasHoldem (2.8MB; 62,576 calls; 4,864 returns; 1 indirect jump) our rewriting engine required less than half a second to rewrite the entire application.

On the other hand, our tests on iOS applications revealed that *MoCFI* raises certain false positives. A remarkable false positive occurs for large jump tables: For theses cases, the compiler calls dedicated switch functions that calculate the address to be used as jump target, where the particular function call is realized via a `BLX` instruction. However, these switch functions never return; instead they use the value of `lr` (loaded via `BLX`) as base pointer to the jump table, and use an indirect jump after the target has been calculated (i.e, a return to the initial call is missing). In order to avoid this false positive, *MoCFI* could be extended if the bytes succeeding the `BLX` are belonging to a jump table. If so, *MoCFI* could enforce a different check in the switch functions: rather than checking the return address, we check if the last indirect jump of the switch function targets an address that is used in the jump table. We aim to integrate additional handling of exceptional cases in our future. In general, one can apply similar policies as mentioned above, or exception handling rules as discussed in [16].

Mitigating Advanced Attacks. In order to demonstrate that *MoCFI* detects advanced attacks that hijack the control-flow of an application, we adopted a return-oriented programming (ROP) attack presented by Iozzo et al. [25] (developed for iOS v2.2.1) to iOS v4.3.1 and extended it in such a way that it bypasses memory randomization on iOS. Specifically, our sample attack exploits a buffer overflow vulnerability and forces the device to beep and vibrate. When protecting the vulnerable application with *MoCFI*, the attack fails and we successfully prevent an exploitation attempt. We implemented the exploit and used techniques similar to *GOT dereferencing* [45, 22] to bypass ASLR on iOS. Since constructing iOS exploits is not the main objective of this paper, we refer the interested reader to Appendix B where we describe our vulnerable program and the payload we used.

8. Related Work

Control-flow (runtime) attacks are a prevalent attack vector since about two decades and a lot of research has been performed to either exploit such vulnerabilities or to find ways to protect against them. In the following, we focus on defense strategies to prevent control-flow attacks and discuss how previous works relates to the approach presented in this paper.

Control-Flow Integrity. The basic principle of monitoring the control-flow of an application in order to enforce a specific security policy has been introduced by Kiriansky et al. in their seminal work on *program shepherding* [29]. This technique allows arbitrary restrictions to be placed on control transfers and code origins, and the authors showed how such an approach can be used to confine a given application. A more fine-grained analysis was presented by Abadi et al., who proposed *Control Flow Integrity* enforcement [1]. We use CFI as the basic technique and show that this principle can be applied on the ARM processor architecture to protect smartphones against control-flow attacks.

Several architectural differences and peculiarities of mobile operating systems complicate our approach and we had to overcome several obstacles. XFI [2] is an extension to CFI that adds further integrity constraints for example on memory and the stack at the cost of a higher performance overhead. The current prototype of *MoCFI* does not implement these additional constraints, but our framework could be extended in the future to also support such constraints.

In contrast to the original CFI work and our *MoCFI*, *Write Integrity Testing* (WIT) [3] also detects non-control-data attacks. This is achieved by interprocedural points-to analysis which outputs the CFG and computes the set of objects that can be written by each instruction in the program. Based on the result of the points-to analysis, WIT assigns a color to each object and each write instruction. WIT enforces write-integrity by only allowing the write operation if the originating instruction and the target object share the same color. As a second line of defense, it also enforces CFI to check if an indirect call targets a valid execution path in the CFG. However, WIT does not prevent return-oriented attacks, because it does not check function returns. Moreover, it requires access to source code. In contrast, *MoCFI* can protect an application against advanced attacks and our tool works directly on the binary level.

HyperSafe [42] protects x86 hypervisors by enforcing hypervisor code integrity and CFI. Similar to *MoCFI*, it instruments indirect branch instructions to validate if their branch target follows a valid execution path in the CFG. However, HyperSafe only validates if the return address is within a set of possible return addresses which has been calculated offline. In contrast to HyperSafe, *MoCFI* enforces fine-grained return address checks, and does not require source code. Moreover, the dynamic nature of smartphone applications, prevents us from calculating return addresses offline.

Native Client (NaCl) [46, 36] provides a sandbox for untrusted native code in web browsers. In particular, NaCl enforces software fault isolation (SFI [41]) and constraints branches to an aligned address. However, this still allows an adversary to subvert the control-flow (as long as the target address is aligned). Moreover, NaCl does not support THUMB code (which is main instruction set on smartphones) and requires recompilation of applications as well.

In a very recent work Zeng et al. [47] showed that CFI combined with static analysis enables the enforcement of efficient data sandboxing. In particular, the presented scheme provides confidentiality of critical memory regions by constraining memory reads to uncritical data regions. This is achieved by placing guard zones before and after the uncritical data area. The solution has been implemented in the LLVM compiler infrastructure (similar to the NaCl compiler [46, 36]) and targets the Intel x86 platform.

Orthogonal Defenses. Many techniques to detect or prevent control-flow attacks have been proposed in the last few years such as for example stack canaries [15], return address stacks [12, 21], and pointer encryption [14]. Such techniques are orthogonal to CFI and focus on specific aspects of exploits. As a result, an attacker might find novel ways to bypass them in order to exploit a given vulnerability.

9. Conclusion and Future Work

In this paper, we focus on the problem of mitigating runtime attacks on modern smartphone platforms. This class of attacks on software is still one of the major threats we need to deal with and we recently saw several runtime attacks against smartphones. We showed how the principle of *control-flow integrity* (CFI) enforcement can be applied on the ARM platform. Our solution tackles several unique challenges of ARM and smartphones operating systems, which we discussed in detail. We solved all challenges and implemented a complete CFI enforcement framework for Apple iOS. Our evaluation shows that we can successfully mitigate even advanced attacks. Moreover, our performance measurements show that *MoCFI* is efficient: it performs well in worst-case scenarios (e.g., computationally intensive algorithms such as quicksort) and does not induce any notable performance overhead when applied to popular iOS applications.

Our current prototype implementation protects the main application binary against control-flow attacks. Therefore, we aim to apply *MoCFI* to shared iOS libraries in our future work, which should be straightforward since there are no conceptional obstacles to overcome. Besides working on a formal analysis along the lines of the original CFI proposal [1], we are currently investigating the possibility of runtime attestation (a trusted computing mechanism to attest the software state of remote platforms) and enhanced application sandboxing based on CFI enforcement.

Acknowledgments

This work has been supported by the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (grant 315-43-02/2-005-WFBO-009), the Federal Ministry of Education and Research (grant 01BY1020 MobWorm), and by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°257243 (TClouds project: `http:/www.tclouds-project.eu`). This work was also partially supported by the ONR under grant N000140911042 and by the National Science Foundation (NSF) under grants CNS-0845559, CNS-0905537, and CNS-0716095.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] M. Abadi, M. Budiu, U. Erlingsson, G. C. Necula, and M. Vrable. XFI: Software Guards for System Address Spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. *IEEE Symposium on Security and Privacy*, 2008.

[4] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 49(14), 1996.

[5] Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), 2001.

[6] Apple Inc. Manual Page of `dyld` - the dynamic link editor. `http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/dyld.1.html`, 2011.

[7] Apple Inc. Manual Page of `mmap` - allocate memory, or map files or devices into memory. `http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/mmap.2.html`, 2011.

[8] ARM Limited. Procedure Call Standard for the ARM Architecture. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf`, 2009.

[9] blexim. Basic Integer Overflows. *Phrack Magazine*, 60(10), 2002.

[10] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[11] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented Programming Without Returns. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[12] T. Chiueh and F.-H. Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[13] comex. `http://www.jailbreakme.com//#`.

[14] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointguardTM: Protecting Pointers From Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*, 2003.

[15] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[16] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[17] J. Duart. Objective-C helper script. `https://github.com/zynamics/objc-helper-plugin-ida`, 2010.

[18] A. Edwards, A. Srivastava, and H. Vo. Vulcan Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[19] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Symposium on Network and Distributed System Security (NDSS)*, 2011.

[20] A. Francillon and C. Castelluccia. Code Injection Attacks on Harvard-Architecture Devices. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[21] M. Frantzen and M. Shuey. StackGhost: Hardware Facilitated Stack Protection. In *USENIX Security Symposium*, 2001.

[22] G. Fresi Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically Returning to Randomized lib(c). In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[23] gera. Advances in Format String Exploitation. *Phrack Magazine*, 59(12), 2002.

[24] R. Hund, T. Holz, and F. C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *USENIX Security Symposium*, 2009.

[25] V. Iozzo and C. Miller. Fun and Games with Mac OS X and iPhone Payloads. In *Black Hat Europe*, 2009.

[26] V. Iozzo and R.-P. Weinmann. PWN2OWN contest. `http://blog.zynamics.com/2010/03/24/ralf-philipp-weinmann-vincenzo-iozzo-own-the-iphone-at-pwn2own/`, 2010.

[27] M. Keith. Android 2.0-2.1 Reverse Shell Exploit, 2010. `http://www.exploit-db.com/exploits/15423/`.

[28] I. King. Will Intel finally crack smartphones? `http://www.businessweek.com/magazine/content/11_25/b4233041946230.htm`, 2011.

[29] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution via Program Shepherding. In *USENIX Security Symposium*, 2002.

[30] T. Kornau. Return Oriented Programming for the ARM Architecture. Master's thesis, Ruhr-University Bochum, 2009.

[31] Microsoft. The `/SAFESEH` compiler flag. `http://msdn.microsoft.com/en-us/library/9a89h429%28v=vs.80%29.aspx`, 2011.

[32] C. Miller and D. D. Zovi. *The Mac Hacker's Handbook*. Wiley Publishing, 2009.

[33] C. Mulliner and C. Miller. Injecting SMS Messages Into Smart Phones for Security Analysis. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[34] Nergal. The Advanced return-into-lib(c) Exploits: PaX Case Study. *Phrack Magazine*, 58(4), 2001.

[35] B. Prince. Security Expert Evades Apple's Mobile Security Measures via iOS Vulnerability. `http://s1.securityweek.com/apple-security-expert-evades-apples-mobile-security-measures-ios-vulnerability`, 2011.

[36] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security Symposium*, 2010.

[37] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[38] H. Shacham, E. jin Goh, N. Modadugu, B. Pfaff, and D. Boneh. On the Effectiveness of Address-space Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.
[39] Solar Designer. "return-to-libc" attack. Bugtraq, 1997.
[40] A. Sotirov and M. Dowd. Bypassing Browser Memory Protections in Windows Vista - Setting back browser security by 10 years. http://www.phreedom.org/research/bypassing-browser-memory-protections/, 2008.
[41] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5), 1993.
[42] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy*, 2010.
[43] R.-P. Weinmann. All Your Baseband Are Belong To Us. hack.lu, 2010. http://2010.hack.lu/archive/2010/Weinmann-All-Your-Baseband-Are-Belong-To-Us-slides.pdf.
[44] S. Winwood and M. Chakravarty. Secure Untrusted Binaries – Provably! In *3rd international workshop on formal aspects in security and trust*, 2006.
[45] R. Wojtczuk. Defeating Solar Designer's Non-executable Stack Patch. http://insecure.org/sploits/non-executable.stack.problems.html, 1998.
[46] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *IEEE Symposium on Security and Privacy*, 2009.
[47] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2011.
[48] D. D. Zovi. Apple iOS Security Evaluation: Vulnerability Analysis and Data Encryption. In *Black Hat USA*, 2011.

APPENDIX

In the following, we describe the implementation of our trampolines, and present a return-oriented programming payload that ASLR on iOS.

A. Trampoline Implementation

Listing 2 and 3 show the implementation of our ARM and THUMB trampoline for all indirect branches (i.e., indirect calls / jumps, and returns). In contrast to the ARM trampoline, the instruction which precedes the (replaced) indirect branch is placed on the top of the THUMB trampoline. The subsequent instructions save the registers on the stack, load the pointer to the patch structure into `r1`, and the pointer to the specific validation routine in `r2`. Furthermore, `sp` is loaded into `r0` so that the invoked validation routine can access the register set of the current execution state. Afterwards, the `BLX` instruction is used to invoke the validation routine. Upon function return, all registers are reset to their original state. Finally, the original branch instruction is executed. Note that we avoid setting `lr` for indirect calls[3] by transforming it to an indirect `BX` jump. Otherwise, the called function would return to our trampoline, and not to the original caller.

```
char customThumbTrampoline[] = {
  0x00,0xbf,0x00,0xbf,  // prev. instruction
  0x2d,0xe9,0xff,0x5f,  // push.w {r0-r12,lr}
  0x68,0x46,            // mov r0,sp
  0x03,0x49,            // ldr.w r1,[pc,12]
  0x03,0x4a,            // ldr.w r2,[pc,12]
  0x90,0x47,            // blx r2
  0xbd,0xe8,0xff,0x5f,  // pop.w {r0-r12,lr}
  0x00,0x00,0x00,0x00,  // orig. instruction
  0x00,0x00,0x00,0x00,  // *patch structure
  0x00,0x00,0x00,0x00   // *validation routine
};
```

Listing 2. THUMB Trampoline

```
ulong customARMTrampoline[] = {
  0xe92d5fff,  // stmfd sp!,{r0-r12,lr}
  0xe1a0000d,  // mov r0,sp
  0xe59f1010,  // ldr r1,[pc,16]
  0xe59f2010,  // ldr r2,[pc,16]
  0xe12fff32,  // blx r2
  0xe50d0004,  // str r0,[sp,-4]
  0xe8bd5fff,  // ldmfd sp!,{r0-r12,lr}
  0x00000000,  // orig. instruction
  0x00000000,  // *patch structure
  0x00000000,  // *validation routine
};
```

Listing 3. ARM Trampoline

Our generic trampolines for direct calls are implemented in ARM (see Listing 4 and 5). Note that this does not raise an interworking problem since direct calls are overwritten with a `BLX` instruction.

```
long ExternalCallTrampoline[] = {
  0xe92d5fff,  // stmfd sp!,{r0-r12,lr}
  0xe1a0000d,  // mov r0,sp
  0xe59f1028,  // ldr r1,[pc,#40] ; -> validate
  0xe12fff31,  // blx r1
  0xe50d0004,  // str r0,[sp,#-4]
  0xe8bd5fff,  // ldmfd sp!,{r0-r12,lr}
  0xe1a0e00f,  // mov lr,pc
  0xe51df03c,  // ldr pc,[sp,#-60];
  0xe92d000f,  // push {r0,r1,r2,r3}
  0xe59f1010,  // ldr r1,[pc,#16] ; -> get_lr
  0xe12fff31,  // blx r1
  0xe1a0e000,  // mov lr,r0
  0xe8bd000f,  // pop {r0,r1,r2,r3}
  0xe12fff1e,  // bx lr
  0x00000000,  // *validation routine
  0x00000000,  // *get_lr
};
```

Listing 4. External Call Trampoline

[3]However, our validation routine correctly sets `lr` by changing the `lr` value that is stored on the stack (because it has been pushed by the second instruction of the THUMB trampoline).

```
ulong InternalCallTrampoline[] = {
  0xe92d5fff, // stmfd   sp!,{r0-r12,lr}
  0xe1a0000d, // mov     r0, sp
  0xe59f100c, // ldr     r1, [pc,#12]
  0xe12fff31, // blx     r1
  0xe50d0004, // str     r0, [sp,#-4]
  0xe8bd5fff, // ldmfd   sp!, {r0-r12,lr}
  0xe51df03c, // ldr     pc, [sp,#-60]
  0x00000000, // *validation routine
};
```

Listing 5. Internal Call Trampoline

The start of both trampolines is similar to the aforementioned custom trampolines: storing all registers, loading the required parameters, and the final call to the validation routine. Our validation routines for direct calls save the return address, check the parameters for Objective C `msgSend` calls, and finally provide the original branch target in `r0`. This value is stored on the stack, and after resetting all registers, loaded into `pc`. As mentioned in Section 6 calls to external (shared libraries) require in our current implementation specific handling: we store the return address on an external shadow stack and change `lr` so that it points to our trampoline. Hence, when the external function returns, it returns to our trampoline, where we invoke the *get_lr* to retrieve the original return address. Note that the *get_lr* function is also invoked when an external library function returns that has been originally invoked via an indirect call. Therefore our validation for indirect calls checks whether the branch target resides in the same code segment or targets a library function. For the latter one we change `lr` in such a way that *get_lr* is invoked after the library function returns.

B. Control-Flow Attacks Against iOS

In the following we describe how we constructed a sample iOS exploit that circumvents memory randomization and let the device beep and vibrate.

The vulnerable program is realized as follows:

```
FILE *sFile;
void foo(char *path, file_length) {
   char buf[8];
   sFile = fopen(path, ''r'');
   fgets(buf, file_length, sFile);
   fclose(sFile);
}
```

The shown *foo()* function simply opens a file, where the file path and length are provided as parameters to the function. Further, via *fgets()* it reads as many characters as specified by the `file_length` parameter, and finally copies them into the local buffer `buf`. However, *fgets()* does not check the bounds of the buffer `buf`. This in turn allows an adversary to divert the control-flow by overflowing the buffer. This can be achieved by providing a file which length's exceeds the buffer's size (here more than 8 Bytes). Hence, the adversary can overwrite the return address of *foo()*, and inject a ROP payload on the stack. In particular, our constructed ROP gadget chain invokes (1) *AudioServicesPlaySystemSound(0x3ea)* to play a sound and vibrate the phone, and (2) *exit(0)* to close the application. However, our ROP exploit will only succeed if it bypasses ASLR.

Bypassing ASLR on iOS. iOS offers two levels of ASLR protection [48]: (1) full ASLR, and (2) ASLR only for shared libraries and the program heap. The former one randomizes each code and data segment of the program. On the other hand, it can only be applied to applications that are compiled as position independent executables (PIE). While built-in applications such as the Safari Browser are compiled as PIEs, third-party applications typically do not support PIE [48]. Hence, for these applications, iOS only randomizes the base addresses of shared libraries, but omits the randomization of the program binary, and dynamic areas such as the stack. Moreover, the iOS linker `dyld` is always loaded at a fixed location. In the following, we focus on applications that do not support PIE. Nevertheless, control-flow attacks can be also launched against PIE applications, if an adversary finds and exploits a memory disclosure vulnerability.

Our target function *AudioServicesPlaySystemSound()* is from the `AudioToolbox` library that iOS randomizes after each device boot for PIE and non-PIE applications. We successfully adopted GOT dereferencing and GOT overwriting [22] techniques which have been recently deployed on Intel x86 (Linux) to resolve an absolute address of a function the adversary wants to execute. In the following, we briefly describe how principles of GOT dereferencing can be applied to ARM and iOS.

GOT dereferencing exploits a common data leakage problem of the *Global Offset Table* (GOT). Typically, the GOT contains references to library function addresses the program aims to use. iOS uses a very similar data structure called indirect symbol table. By dereferencing a single entry of the indirect symbol table, we can obtain an absolute (i.e., runtime) address of a function, which in turn allows us to calculate the randomization offset by subtracting the static address of the same function. For our specific attack, we read the absolute address of *fgets()* and subtract its static address. We store the calculated offset value into the indirect symbol table entry for *fgets()*, from where it can be loaded each time the offset is needed. We calculate the absolute address of *AudioServicesPlaySystemSound()* by adding the randomization offset to the static address of this function, and redirect the control-flow to the computed address. Note that we use the non-randomized code base of the static `dyld` library to compute the randomization offset

and to resolve absolute addresses of functions (or instruction sequences) from randomized libraries.

Payload and Instruction Sequences. To construct our malicious program, we leverage six different instruction sequences, while several of them are used more than once. The execution order of the instructions sequences is shown in Listing 6.

```
0x2fe16a66: ldma sp!, {r7, lr}
0x2fe16a6a: add sp, #12
0x2fe16a6c: bx lr

0x2fe06528: pop {r1,r3,r5,r7,pc}

0x2fe1e4c8: pop {r0,r4,r5,r7,pc}

0x2fe0efc2: ldr r1, [r1, #0]
0x2fe0efc4: adds r0,r0,r1
0x2fe0efc6: bx lr

0x2fe06528: pop {r1,r3,r5,r7,pc}

0x2fe0f0e4: str r0, [r1, #20]
0x2fe0f0e6: bx lr

0x2fe06528: pop {r1,r3,r5,r7,pc}

0x2fe1e4c8: pop {r0,r4,r5,r7,pc}

0x2fe0efc2: ldr r1, [r1, #0]
0x2fe0efc4: adds r0,r0,r1
0x2fe0efc6: bx lr

0x2fe06528: pop {r1,r3,r5,r7,pc}

0x2fe0ec40: mov r12, r0
0x2fe0ec44: pop {r0,r1,r2,r3,r7,lr}
0x2fe0ec48: add sp, sp, #8
0x2fe0ec4c: bx r12

0x2fe1e4c8: pop {r0,r4,r5,r7,pc}
```

Listing 6. Instruction Sequences

```
0000: 41 41 41 41  41 41 41 41  30 30 30 30
000C: 67 ba e1 2f  30 30 30 30  29 65 e0 2f
0018: 41 41 41 41  41 41 41 41  41 41 41 41
0024: 10 30 00 00  41 41 41 41  41 41 41 41
0030: 30 30 30 30  c8 e4 e1 2f  1f 61 0c cc
003C: 41 41 41 41  41 41 41 41  30 30 30 30
0048: c3 ef e0 2f  fc 2f 00 00  41 41 41 41
0054: 41 41 41 41  30 30 30 30  e5 f0 e0 2f
0060: 10 30 00 00  41 41 41 41  41 41 41 41
006C: 30 30 30 30  c8 e4 e1 2f  cd ca 8a 33
0078: 41 41 41 41  41 41 41 41  30 30 30 30
0084: c3 ef e0 2f  41 41 41 41  41 41 41 41
0090: 41 41 41 41  30 30 30 30  40 ec e0 2f
009C: ea 03 00 00  41 41 41 41  41 41 41 41
00A8: 41 41 41 41  30 30 30 30  c8 e4 e1 2f
00B4: 30 30 30 30  41 41 41 41  00 00 00 00
00C0: 41 41 41 41  41 41 41 41  30 30 30 30
00CC: dc 2f 00 00  30 30 30 30
```

Listing 7. Payload

Our payload is shown in Listing 7. The first two words of the payload fill the buffer `buf`. The next two words are popped from the stack into `r7` and `pc` upon return of *foo()*. Specifically, our exploit overwrites the return address with `0x2fe16a67` to redirect execution to the first instruction sequence. The following five addresses point to the subsequent sequences: `0x2fe06529`, `0x2fe1e4c8`, `0x2fe0efc3`, `0x2fe0f0e5`, and `0x2fe0ec40`. Note that code sequences compiled in THUMB mode have to be addressed by an odd value (+1).

The address `0x33a8cacd` is the static address of *AudioServicesPlaySystemSound()*. Further, the address `0x3010` points to the absolute address of *fgets()* stored in the indirect symbol table of the vulnerable program. The value `0xcc0c611f` is an inversion of the static address of *fgets()* `0x33f39ee0`. `0x03ea` and `0x00` are parameters for the functions *AudioServicesPlaySystemSound()* and *exit()*, respectively. Finally, `0x2fdc` is the address of the indirect symbol table entry of *exit()*. We use `0x41414141` and `0x30303030` as pattern bytes to compensate the side effects of our invoked sequences.