POSTER: Control-Flow Integrity for 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]Technische Universität Darmstadt, Germany [2]Fraunhofer SIT, Darmstadt, Germany
[3]University of California, Santa Barbara, USA [4]Ruhr-Universität Bochum, Germany

ABSTRACT

Despite extensive research over the last two decades, runtime attacks on software are still prevalent. Recently, smartphones, of which millions are in use today, have become an attractive target for adversaries. However, existing solutions are either ad-hoc or limited in their effectiveness.

In this poster, we present a general countermeasure against runtime attacks on smartphone platforms. Our approach makes use of *control-flow integrity* (CFI), and tackles unique challenges of the ARM architecture and smartphone platforms. Our framework and implementation is efficient, since it requires no access to source code, performs CFI enforcement *on-the-fly* during runtime, and is compatible to memory randomization and code signing/encryption. We chose Apple iPhone for our reference implementation, because it has become an attractive target for runtime attacks. Our performance evaluation on a real iOS device demonstrates that our implementation does not induce any notable overhead when applied to popular iOS applications.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

1. INTRODUCTION

Although runtime attacks on software are known for about two decades, they are still one of the major threats we need to deal with today. Such attacks compromise the control-flow of a vulnerable application during runtime based on diverse techniques (e.g., stack smashing or heap overflows). Many current systems offer a large attack surface, because they still deploy large amounts of native code implemented in unsafe languages such as C/C++. In particular, modern smartphones have recently become an appealing attack target (e.g., [11, 8]).

A general approach to mitigate runtime 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 legitimate paths determined in advance. If an adversary hijacks the control-flow, CFI enforcement can detect this divergence and prevent the attack.

In contrast to many proposed ad-hoc solutions, CFI does not only consider a specific attack, but instead provides a general solution against control-flow attacks. Surprisingly, and to the best of our knowledge, there exist no published CFI approach for smartphone platforms.

In this poster, we present the design of a CFI enforcement framework for smartphone platforms. Specifically, we focus on the ARM architecture since it is the standard platform for smartphones. The implementation of CFI on ARM is often more involved than on desktop PCs due to the following subtle architectural differences: (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 iPhone (iOS) for our reference implementation because of three challenging issues: First, iOS is a popular target for runtime attacks due to its use of the Objective-C programming language. Second, iOS is closed-source: We can neither change the actual operating system nor can we access the source code of an application. Third, applications are encrypted and signed by default.

Contribution.

To the best of our knowledge, we present the first general CFI enforcement framework for smartphone platforms. Our solution tackles unique challenges of smartphones, does not require access to source code, and can be transparently enabled for individual applications. Moreover, we launched popular iOS applications as well as computationally intensive algorithms under the protection of our CFI framework, and can show that our implementation efficiently handles them. To this end, we first implemented a system to recover the control-flow graph (CFG) of a given iOS application in binary format. 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 completely compatible to memory randomization and code signing.

2. PROBLEM DESCRIPTION

Figure 1 depicts a sample *control-flow graph* (CFG) of an application. Basically, the CFG represents valid execution paths of a program. It consists of *basic blocks* (BBLs), instruction sequences with a single entry and a exit instruction (e.g., return, call, or jump), where the exit instruction

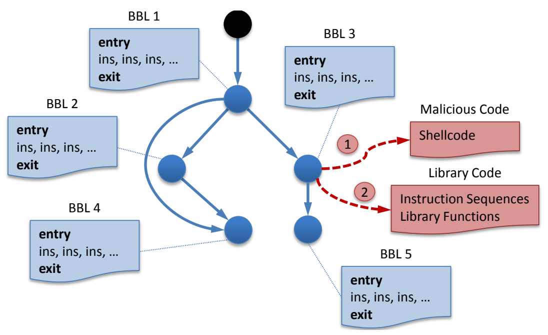

Figure 1: Schematic overview of control-flow attacks

enables the transition from one BBL to another BBL. 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: (1) a *code injection attack*, and (2) a *code reuse attack*. 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 [2]. However, modern operating systems (such as iOS) enforce the $W \oplus X$ (Writable xor Executable) security model that prevents an adversary from executing injected code. On the other hand, code-reuse attacks such as return-oriented programming [3, 7, 9, 10] bypass $W \oplus X$ by redirecting execution to code already residing in the program's memory space.

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 [11], to open a remote reverse shell [8], or to launch a jailbreak [5]. Unfortunately, there exist no general countermeasure to defeat such attacks on smartphones.

3. DESIGN OF OUR CFI FRAMEWORK

In this section we introduce the high-level idea of our CFI framework for smartphone platforms. Our general architecture is shown in Figure 2. Although the depicted design applies in general to all CFI solutions, our design requires a number of changes, mainly due to (1) the architectural differences between ARM and Intel x86, (2) the missing binary rewriter and automatic graph generation for ARM, and (3) the specifics of smartphone operating systems.

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 iOS binary file: we first decrypt and disassemble the binary and then extract the control-flow graph (CFG) and all meta information necessary for rewriting a particular iOS binary in the enforcement phase. We monitor the application at *runtime* by applying our *CFIKit* shared library that rewrites the binary at load-time and enforces control-flow restrictions while the application executes.

3.1 Control-Flow Graph Generation

Since no binary instrumentation framework for ARM exist we developed own techniques to accurately generate the CFG. First, we disassemble the application binary (step 1). In our case, this is impeded by the fact that iOS executables are encrypted. We thus obtain the unencrypted code of a binary through *process dumping* [6]. The decrypted and disassembled iOS binary is afterwards persistently stored (step 2). Subsequently, we generate the CFG and the rewriting information for the runtime components (step 3 and 4). The latter contains information on where and how the rewriting engine should dispatch the application's code to its validation routines. To generate the CFG we developed static tools that calculate the targets of indirect jumps and calls.

3.2 Load-Time Module: Binary Rewriting

The binary rewriting engine is responsible for binding additional code to the binary (step 5) that checks if the application follows a valid execution path of the CFG. Typically, one replaces all branches 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.

Due to the limited possibilities to change iOS binaries (code signing) and the missing full binary rewriter, we opted for the following approach: Based on the extracted rewriting information we replace all relevant branches with a single instruction, the so-called *trampoline instruction*. The trampoline instruction redirects execution to our *CFIKit* library.

3.3 Runtime Module: CFI Enforcement

The key insight of CFI is the realization of control-flow validation routines. These routines have to validate the target of every branch (step 6) 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 require special handling because return addresses are dynamic and cannot be predicted ahead of time. To address this issue, *CFIKit* reuses the concept of shadow stacks that hold valid copies of return addresses [4], while the return addresses are pushed onto the shadow stacks when function calls occur.

A very challenging issue on iOS are method calls to an Objective-C object. These are resolved to a call to the generic message handling function called `objc_msgSend`. The name of the actual method (called *selector*) to be called is given as a parameter. While the traditional CFI approach omits the handling of direct function calls, our *CFIKit* has to consider direct calls to Objective-C objects. 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 [6] and use it to generate call graph information for `objc_msgSend` calls.

4. IMPLEMENTATION

Our prototype implementation targets iOS 4.3.2. We developed the static analysis tools with the IDC scripting language featured by the well-known disassembler IDA Pro 6.0. Moreover, we used Xcode 4 to develop the *CFIKit* library.

Our IDC scripts extract rewriting information and generate the CFG, and store both in the application bundle. To force the loading of *CFIKit* into every application started though the touchscreen, we set the environment variable `DYLD_INSERT_LIBRARIES` for the *SpringBoard* process. This

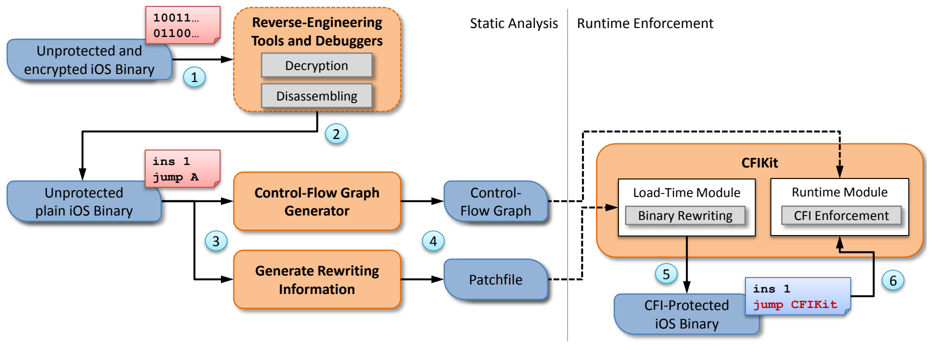

Figure 2: Control-flow integrity for iOS applications

ensures that the loader always loads *CFIKit* before any other dependency of the actual program binary. Note that our solution *only* requires a jailbreak for performing these two tasks. Hence, our solution can be easily integrated into Apple's software development cycle.

Once *CFIKit* has been initialized, it rewrites the application binary *on-the-fly*. It mainly replaces branch instructions with trampoline instructions that target a piece of optimized assembler code, namely the trampoline, that is used as a bridge between the application and our *CFIKit* library. Specifically, we allocate dedicated trampolines for each relevant branch in the program, where each trampoline (1) saves the current execution state, (2) invokes the appropriate *CFIKit* validation routine, and (3) resets the execution state and issues the original branch. Because of step (3), we guarantee that all registers are loaded correctly, even if the branch loads several registers as a side-effect. Moreover, depending on the replaced branch instruction, we allocate a THUMB or ARM trampoline to ensure the correct interworking between the two instruction sets.

Note that we in particular faced the following challenge: most parts of the program code are compiled in 16 Bit THUMB mode. Nevertheless, direct branches require 32 Bits in THUMB mode. Hence, a 16 Bit indirect branch has to be replaced with a 32 Bit trampoline instruction. To solve this issue, we replace 32 Bits in the program text (thereby overwriting 2 Thumb instructions). To preserve the program's semantics, we execute the instruction that precedes the branch at the beginning of our trampolines.

However, this approach only works if the mentioned instruction does not reference the program counter or is also a branch. In such scenarios, we use an entirely different approach: upon initialization, we register an iOS exception handler for illegal instructions. The trampoline instruction is then simply an illegal instruction that will trigger our exception handler. Since this technique induces additional performance overhead we only use it for exceptional cases.

Evaluation.

We applied *CFIKit* to a quicksort program that frequently asks for a control-flow check. Even in this worst-case scenario *CFIKit* performs quite well and needs only 81ms to run a quicksort for $n = 10,000$ (see Table 1). Moreover, we successfully applied *CFIKit* to the iOS Facebook application code (2.3MB containing more than 33,647 function calls and 5,988 returns) and did not notice any performance penalties while executing the application. Further, our rewriting engine only required 0.5s to rewrite the entire application.

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

Table 1: Measurement results for quicksort

5. CONCLUSION

In this poster, we introduced a general countermeasure against runtime attacks on smartphone platforms. We presented a complete control-flow integrity (CFI) framework for the closed-source Apple iOS. Our solution requires no access to source code, rewrites binaries on-the-fly, and performs control-flow checks at runtime. Our performance measurements demonstrate that our framework is efficient for computationally intensive tasks and popular iOS applications.

6. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *ACM CCS*, 2005.

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

[3] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented Programming Without Returns. In *ACM CCS*, 2010.

[4] T. Chiueh and F.-H. Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *ICDCS*, 2001.

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

[6] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS*, 2011.

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

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

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

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

[11] R.-P. Weinmann and V. Iozzo. Ralf-Philipp Weinmann & Vincenzo Iozzo own the iPhone at PWN2OWN, 2010.