CFI Goes Mobile: 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], and 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, control-flow (or 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 paper, we present a general countermeasure against control-flow attacks on smartphone platforms. Our approach makes use of *control-flow integrity* (CFI), and tackles unique challenges of the ARM architecture and smartphone platforms (e.g., application encryption and signing, closed-source OS). 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 (e.g., ASLR) and code signing/encryption. We chose Apple iPhone for our reference implementation, because it has become an attractive target for control-flow attacks due to its wide spread deployment of native code. Our performance evaluation on a real iOS device demonstrates that our implementation does not induce any notable overhead when applied to popular iOS applications.

Keywords: Control-Flow Integrity, ARM, Software Security

1 Introduction

Although control-flow 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++. The target of attacks range from desktop PCs to embedded systems. In particular, modern smartphones like Apple's iPhone and Google's Android have recently become an appealing attack target (e.g., [12, 13]).

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 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 paper, 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 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 iPhone for our reference implementation because of three challenging issues: First, the iPhone platform is a popular target for control-flow attacks due to its use of the Objective-C programming language. Second, iOS is closed-source: We can neither change the operating system nor can we access the application source code. Third, applications are encrypted and signed by default.

To the best of our knowledge, we provide the first 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 (such as quicksort) 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/encryption.

2 Background

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

2.1 ARM Architecture

ARM features a 32 bit processor and 16 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. In contrast to Intel x86, machine instructions are allowed to directly operate on the program counter `pc` (`%eip` on x86). For instance, it is possible to load `pc` with a new value from the stack.

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 bits long 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 [5]. In general, a function can be called via a `BL` (**B**ranch with **L**ink) or `BLX` (**B**ranch with **L**ink and e**X**change) instruction, where both instructions require a `pc`-relative offset as branch target (for direct calls). The major difference between `BL` and `BLX` is that `BLX` additionally allows indirect calls (i.e., the branch target is stored in a register), and the exchange (often referred to as *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`. Since ARM does not feature a dedicated return instruction, *any* instruction able to load 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 return does not only enforce the return to the caller, but also loads several registers within a single instruction. For instance, many returns are realized by a compiler as `POP {R4,R7,PC}`. This `POP` instruction will load `R4` and `R7` with new values from the stack, and also pop the return address to `pc`. This unique characteristic significantly complicates control-flow analysis on ARM.

2.2 Apple iOS

Apple iOS is a closed and proprietary operating system that is mainly based on Mac OS X and 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. This reduces the attack surface for malicious software. Furthermore, Apple only signs applications after inspecting the application code. However, Apple provides no information how code inspection is enforced. Moreover, Apple has only access to the application binary (not to the 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]. However, advanced attacks such

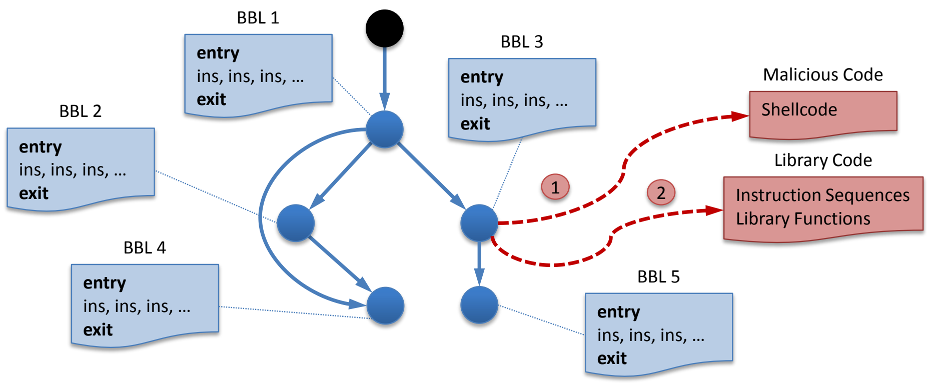

Fig. 1. Schematic overview of control-flow attacks

as return-oriented programming (ROP) [17] bypass $W \oplus X$ by combining only existing code pieces of a vulnerable program and dynamic libraries.

To detect stack overflows, iOS deploys the so-called Stack-Smashing Protector (SSP). Basically, SSP uses *canaries*, i.e., guard values that are placed between local variables and control-flow information, to detect stack smashing attacks. A canary corruption will abort the execution of the program, thus preventing the execution of a malicious payload. 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. Finally, SSP can not provide protection against heap overflows or any other kind of 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 the stack and the heap, thus preventing an adversary from guessing the location of injected code (or useful library sequences executed in a ROP attack). However, existing randomization realizations are often vulnerable to brute-forcing [18] or leak sensitive information about the memory layout [19].

3 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 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 [4]. 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 [6, 10, 15, 17] 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 [12], to open a remote reverse shell [13], or to launch a jailbreak [8]. Unfortunately, there exist no general countermeasure to defeat such attacks on smartphones.

4 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.

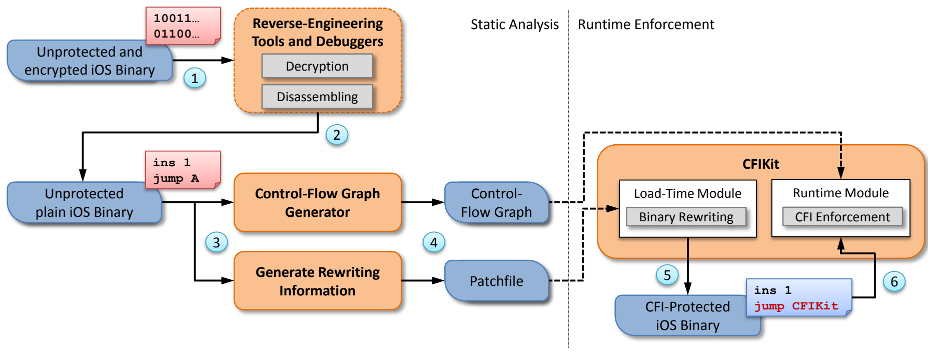

Fig. 2. Control-flow integrity for iOS applications

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.

4.1 Control-Flow Graph Generation

Since no binary instrumentation framework for ARM exists 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* [9]. 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.

4.2 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 execution 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.

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.

4.3 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 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 of copies of return addresses [7], while the return addresses are pushed onto the shadow stacks when a function call occurs.

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 [9] and use it to generate call graph information for `objc_msgSend` calls.

5 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 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 *CFIKit* 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 *CFIKit* library.

Our IDC scripts extract rewriting information and generate the CFG, and store both in the application bundle. By bundling this information with the application, we can protect its integrity, as all application bundle contents are code-signed. Note that our scripts have to be run only once after compilation and can be integrated as an additional step in the deployment phase of a typical iOS application. 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 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 the two aforementioned 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. The trampoline instruction is a ARM branch (`B`) instruction and targets a small piece of optimized assembler code, namely the trampoline itself, that is used as a bridge between the application to be protected 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 (e.g., storing relevant registers), (2) invokes the appropriate *CFIKit* validation routine, and (3) resets the execution state and issues the original branch (which is copied at the end of each trampoline). 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. Another reason for using dedicated trampolines is the following: when using a single trampoline we would need to find out from where the trampoline has been called to issue the original branch and to correctly load the involved registers (for the case the branch loads registers as a side-effect) after a successful CFI validation. Obviously, one could use the `BLX` instruction (see Section 2.1) as trampoline instruction, and calculate the origin by simply reading the lr register. However, note that this approach will destroy the value of lr which may be needed if a return is realized over a `BX lr` instruction. Moreover, one has to record the specific registers to be loaded for each return instruction.

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 indirectly 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 arbitrary illegal instruction that will trigger this 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

To check the effectiveness of our *CFIKit*, we constructed a sample vulnerable application and exploit payload based on the attack presented by Iozzo and Miller [11] (developed for iOS v2.2.1). Specifically, we extended the exploit to bypass the latest security features on iOS such as ASLR. Our demo attack exploits the `fgets()` function and applies principles of return-oriented programming to subvert the control-flow of a vulnerable application and to force a device to beep and vibrate. When protecting the vulnerable application with *CFIKit*, the attack fails and we successfully prevent an exploitation attempt.

6 Related Work

Control-flow 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.

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* [14]. 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 *CFIKit* does not implement these additional constraints, but our framework could be extended in the future to also support such constraints. Again, the adoption to the ARM processor architecture is a challenge.

In contrast to the original CFI paper and our *CFIKit*, *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 have 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, *CFIKit* can protect an application against advanced attacks and our tool works directly on the binary level.

HyperSafe [21] protects x86 hypervisors by enforcing hypervisor code integrity and CFI. Similar to CFIKit, 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, *CFIKit* 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) [22, 16] provides a sandbox for untrusted native code in web browsers. In particular, NaCl enforces software fault isolation (SFI [20]) and lightweight CFI (i.e., the target of an indirect branch is aligned). 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 the standard instruction set on smartphones) and requires access to source code as well.

7 Conclusion

In this paper, we introduced a general countermeasure against control-flow attacks on smartphone platforms. In particular, 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 evaluation demonstrates that we can successfully mitigate advanced attacks utilizing return-oriented programming. Moreover, our performance measurements demonstrate that our framework 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 (such as Facebook). In our future work we aim to enforce security policies on top of CFI, e.g., attestation, a concept known from the Trusted Computing area, that allows remote verifiers to attest platforms. In particular, we are interested in runtime attestation, a mechanism that states if applications have been compromised by a control-flow (or runtime) attack.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-Flow Integrity: Principles, Implementations, and Applications. In: ACM Conference on Computer and Communications Security (CCS) (2005)
2. Abadi, M., Budiu, M., Erlingsson, U., Necula, G.C., Vrable, M.: XFI: Software Guards for System Address Spaces. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2006)
3. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: 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. ARM Limited: Procedure Call Standard for the ARM Architecture. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf` (2009)
6. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented Programming Without Returns. In: ACM Conference on Computer and Communications Security (CCS) (2010)
7. Chiueh, T., Hsu, F.H.: RAD: A Compile-Time Solution to Buffer Overflow Attacks. In: International Conference on Distributed Computing Systems (ICDCS) (2001)
8. comex: `http://www.jailbreakme.com//#`
9. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: PiOS: Detecting Privacy Leaks in iOS Applications. In: Symposium on Network and Distributed System Security (NDSS) (2011)

10. Hund, R., Holz, T., Freiling, F.C.: Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In: USENIX Security Symposium (2009)
11. Iozzo, V., Miller, C.: Fun and games with Mac OS X and iPhone payloads. In: Black Hat Europe (2009)
12. Iozzo, V., Weinmann, R.: PWN2OWN contest. `http://blog.zynamics.com/2010/03/24/ralf-philipp-weinmann-vincenzo-iozzo-own-the-iphone-at-pwn2own/` (2010)
13. Keith, M.: Android 2.0-2.1 Reverse Shell Exploit (2010), `http://www.exploit-db.com/exploits/15423/`
14. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure Execution via Program Shepherding. In: USENIX Security Symposium (2002)
15. Kornau, T.: Return Oriented Programming for the ARM Architecture. Master's thesis, Ruhr-University Bochum (2009)
16. Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., Chen, B.: Adapting Software Fault Isolation to Contemporary CPU Architectures. In: USENIX Security Symposium (2010)
17. Shacham, H.: 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)
18. Shacham, H., jin Goh, E., Modadugu, N., Pfaff, B., Boneh, D.: On the Effectiveness of Address-space Randomization. In: ACM Conference on Computer and Communications Security (CCS) (2004)
19. Sotirov, A., Dowd, M.: Bypassing Browser Memory Protections in Windows Vista. `http://www.phreedom.org/research/bypassing-browser-memory-protections/` (2008)
20. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient Software-based Fault Isolation. ACM SIGOPS Operating Systems Review 27(5), 203–216 (1993)
21. Wang, Z., Jiang, X.: HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In: IEEE Symposium on Security and Privacy (2010)
22. Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native Client: A Sandbox for Portable, Untrusted x86 Native Code. IEEE Symposium on Security and Privacy (2009)