Control-Flow Restrictor: Compiler-based CFI for iOS

Jannik Pewny and Thorsten Holz
Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
{firstname.lastname}@rub.de

ABSTRACT

Runtime attacks that exploit software vulnerabilities are still an important concern nowadays. Even smartphone operating systems such as Apple's iOS are affected by such attacks since the system is implemented in Objective-C, a programming language that enables attacks such as buffer overflows. As a generic protection technique against a whole class of attacks, *control-flow integrity* (CFI) offers some interesting properties. Recent work demonstrated that CFI can be implemented on iOS by patching the binary during the loading process and adding an instrumentation layer that enforces CFI during runtime. However, this approach is of little practical value since it requires a jailbroken device, which hinders wide employment. Furthermore, binary patching has a certain performance impact.

In this paper, we show how CFI can be implemented directly within a compiler, making the approach widely deployable on all kinds of iOS devices. We extend the LLVM compiler and add our CFI enforcement approach during the compilation phase of a given app. An empirical evaluation shows that the size and performance overhead is reasonable.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Control-Flow Integrity, Compiler, iOS, ARM

1. INTRODUCTION

An important attack vector that is still very common are runtime attacks that take advantage of a programming mistake in a given program. About 40% of recently reported attacks are based on manipulation of memory regions [2].

This includes memory corruption attacks like buffer overflows, stack overflows, heap overflows, integer overflows and format string attacks [3, 5, 14, 17, 37]. One of the worst consequences of such an attack is that an attacker can execute arbitrary code, which enables her to take full control of a given system. The manipulation of memory regions is the underlying root cause for the attacks and is worth prohibiting on its own [2]. However, the immediate cause of arbitrary code execution is the diversion of the control-flow.

There are lots of mitigation techniques against such attacks [19, 25, 37, 40, 41, 43]. Unfortunately, most of them can be circumvented in one way or the other, which makes more pervasive measures necessary. Proving certain properties statically (e.g., [35, 44]) is a very powerful approach, but is usually difficult to apply in practice. Furthermore, there are legacy code bases we need to care about in order to protect existing code against such attacks. An approach to ensure that the control-flow integrity (CFI) is not violated in presence of memory manipulations was presented by Abadi et al [1]. The basic idea is to statically determine all allowed control-flow transfers before runtime and forbid the execution of all others at runtime, prohibiting an attacker from jumping to arbitrary memory locations.

Since even smartphone operating systems such as Apple's iOS are affected by memory corruption attacks, we need protection mechanisms for such devices. Smartphones are becoming more widely used [13] and hold both sensitive and private information. This makes them high-value targets for an attacker, which means that the prevention of arbitrary code execution is especially important on these devices. MoCFI [20] provides an interesting approach for CFI enforcement on mobile devices. The authors analyze the binary of an app to derive the control-flow graph (CFG). At load time, they insert trampolines into a runtime component before each jump. This runtime component then checks each jumps target against the CFG. To the best of our knowledge, MoCFI is the only thorough CFI enforcement solution for iOS, which is one of the two most widely used smartphone operating systems on the market.

However, their approach cannot easily be used in practice. One the one hand, each app is digitally signed, which demands the insertion of property checking code to occur between load time and runtime. As a result, a jailbroken device [28] is needed, which severely limits the practical applicability of the approach. On the other hand, no mature binary rewriting engine for ARM [11], the processor architecture of iOS devices, exists. Therefore, the authors have to insert trampolines instead of the full checks, which is naturally less efficient and requires an additional runtime component.

Instead of analyzing the binary, one can work directly on the source code of the app. Such a compiler-based approach [46,48] has several advantages in practice. First, the deployment of the app is much simpler and compatible with the concept of the AppStore and stock phones. Second, no runtime component is required, because no manipulations of the program after load time are needed. This makes a modified OS or a jailbreak unnecessary, enabling a wide deployment. Third, it is more efficient, as neither trampolines add additional runtime nor hot-patching is carried out. A compiler-based approach allows using the semantic information provided by the programming language instead of reverse engineering the likely meaning of a construct found in the binary. The programmer is also nearby, who should know best which paths should be allowed or forbidden. Of course, a compiler-based approach requires access to the source code. This means that we cannot apply it to the kernel, its shared libraries or the browser, because iOS is closed source. It is also not in our scope to provide an infrastructure, where it is ensured that every app has enforced CFI.

In this paper, we present CFR (short for Control-Flow Restrictor), a compiler-based CFI enforcement solution for iOS. During compilation, we generate the control-flow graph of the app. We then enforce CFI by inserting a little code fragment (i.e., three assembler instructions) before each jump. This allows to check at runtime, if the target of the jump accords to the CFG of the program. Empirical evaluations demonstrate that our solution is efficient, both in additional size and runtime overhead.

In summary, the contributions of this paper are:

- We perform a theoretical analysis of CFI enforcement applicable to ARM.
- We implemented a compiler-based CFI enforcement tool for iOS that overcomes limitations of state-of-the-art approaches.
- With several benchmarks, we demonstrate the practical viability of the proposed tool empirically.

2. BACKGROUND

To provide the necessary background, we first describe the special situation a CFI enforcement tool like CFR is confronted with when it is supposed to work for iOS.

2.1 Attacker Model

In the following, we introduce the attacker model used throughout this paper. Our attacker is not able to compromise the program at compile time or load time. At runtime, she cannot alter code memory or registers without utilizing the instructions of the program. However, she is able to read the code memory and to read and write the data memory, including stack and heap, of the attacked program. She also controls all input streams of the program.

This models the pervasive presence of exploitable memory corruption bugs, which makes the attacker quite powerful. However, it is implied that she can influence neither kernel nor hardware immediately. This model is essentially the same as the one used by Abadi et al. [1].

2.2 Apple Ecosystem

Some of Apple's policies for apps in the AppStore play into the hands of security [7], e.g., every app is encrypted and code-signed. This ensures that an attacker cannot manipulate the app's code between deployment and runtime. It also means that CFI enforcement tools, which insert binary code into an app, either require changes to the OS itself or a jailbroken device, both of which hinder widespread adoption. Furthermore, the $W \oplus X$ policy states that memory can be either writable or executable, but not both at the same time. This means that the injection of code at runtime is much harder for an attacker. Only a few internal iOS libraries are allowed to add code at runtime. If another app trying to do this would make it into the AppStore, it would be terminated by the iOS security sandbox when jumping into dynamically generated code. Together with the $W \oplus X$ policy, this ensures that the code available for execution does not change. However, sandboxed systems benefit from CFI, as additional CFI hinders app-internal control-flow attacks and sandbox-escape attacks significantly.

2.3 Background on ARM

On ARM processors, two aspects favor a compiler-based solution compared to an approach based on binary rewriting. First, the program counter can be used as a general purpose register. Thus, many instructions can target it and thereby influence the control-flow. This heavily affects binary analysis-based tools, because there are different implementations of jumps to consider. In contrast, it matters only little for compiler-based tools, because the implementation of a jump can be chosen by the tool. The second aspect is the presence of different instruction sets, which can even be switched at runtime. Again, this complicates matters for binary analysis-based tools. In contrast, compiler-based tools can simply pick an instruction set and stick to it. Also, since instructions in the basic ARM instruction set are all 32-Bit wide, all jump destinations are 4-Byte aligned by the processor. This prohibits many ROP gadgets [39].

2.3.1 *Types Of Jumps*

We use the term *jump* in its broadest meaning: Any instruction that does not necessarily transfer the control-flow to the immediately following instruction is considered a jump. There are several different types of jumps as explained in the following.

Constant jumps.

Jumps with only a single, constant destination are denoted constant jumps. In C-like languages, they correspond to the statement `goto LABEL;`, but they are also used to implement various constructs, e.g., `if/else` switches and loops. Since their destination is known at compile time, they are usually implemented without referring to data memory and are therefore not manipulable in our attack model.

Indirect jumps.

If the destination of a `goto` is computed at runtime (usually with referring to data memory), the situation is different: The attacker can manipulate the jump and, therefore, the control-flow. Such *indirect jumps* could be constructed by using `if/else` switches and constant jumps, but it is usually more efficient to use actual indirect jumps. Also `switch/case` is frequently implemented with indirect jumps.

Fortunately, the destinations of such a jump have to be defined at compile time in the programming languages C and Objective-C [9] and also in the compiler-framework LLVM [30, 33]. Therefore, one can easily generate a whitelist of valid destinations and check against it at runtime.

Constant function calls.

When a function is called in C-like languages without any indirection due to `struct`, `class` or function pointers, the destination of the call is unambiguous at compile time. The actual jump can be implemented without using data memory and is therefore not manipulable. Yet, it is important to track these calls, because they determine valid destinations for `return`-instructions in the called function.

Indirect function calls.

Indirect function calls use some kind of function pointer. Those are regularly stored in data memory and can therefore be manipulated. The corresponding attack is usually called a *pointer subterfuge*. According to the C standard, any function that has the same signature as the function pointer is a valid target. This means that such a call usually has many semantically valid destinations. Similar to a direct function call, an indirect function call defines a valid destination for each `return` in the called function.

Returns.

The counterpart of the function call is the `return` instruction. As known from the exploitation of different kinds of runtime attacks, the return address is saved in data memory, where it can be manipulated. Each valid destination is implicitly defined by the function calls, which can call the `return`'s function.

Objective-C – objc_MsgSend.

While the former jumps are also present in C, this a specialty of Objective-C, which is the language of the iOS runtime and the language of choice for iOS apps [8]. Therefore, it is also the source language we have to consider for a CFI enforcement tool for iOS. The object-orientation of Objective-C uses a scheme of sender, message and receiver to call the methods of an object. The pendant to a method call is sending a message with method name and parameters to the object.

In Objective-C, a method call `[obj m_X:1 Y:2];` represents a call to an object `obj` with a method named `m_X`, which takes the parameters `X` and `Y`. In this case, their values are 1 and 2. Internally, this []-notation is converted to `objc_MsgSend(obj, m_X:Y:, 1, 2);`. The value `m_X:Y:` is defined by the concatenation of method name and parameter names. The compiler creates a single constant string for each such value. The address of such a string is called a selector [9] (Chapter Selectors). To determine which function to call, the runtime environment can compare the selectors instead of the full strings. For optimization purposes, the runtime environment does not return from the `objc_MsgSend`-function, but directly jumps to the specified method. Therefore, the function pointer is not known to the program itself, but only to the runtime. Instead of checking the function pointer, one has to check against the selector, which, fortunately, cannot be altered, as they are in code-memory, and are the second parameter of the `objc_MsgSend`-function. The ARM calling convention states that the first three parameters are passed in registers [11].

There are a few things complicating `objc_MsgSend`-jumps:

- Different `objc_MsgSend`-functions: There are variations for sending messages to a class or superclass or for different return types. Also, they have a variable number of arguments (`va_list`) and must be casted.

- Dynamic behavior and ambiguity: Objective-C provides various ways for behavior, which can or shall not be determined at compile time.

  – Adding Methods: Methods can be added at runtime, changing the interface of a class.
  – Polymorphism: If a class does not have a handler for a selector it is called with, its special `forwardInvocation` method is called. If this method does not respond to the selector, the superclass is called transparently.
  – Method Swizzling: The function that gets called for a specific selector can be exchanged with another function at runtime.
  – Adding Selectors: Selectors can be provided at runtime (e.g., with the method `NSSelectorFromString`).

- Indirection: Analogue to indirect function calls, selectors can be stored in variables, which can later be passed to the `performSelector:`-function. Also, storing and editing whole messages is possible via the `NSInvocation`-class.

2.3.2 Not Supported Jumps

Several types of jumps cannot be handled easily with our compiler-based approach as we discuss in the following.

Exceptions.

Exceptions are designed as a special way to deal with errors [9] (Chapter Exceptions). In contrast to a returned error code, an exception does not need to be handled by the immediately calling function. Instead, any higher instance in the call tree might do so. The locations of throwing and catching are semantically defined and accessible in LLVM, but iOS implements them with a `setjmp()/longjmp()`-mechanism [36]. `setjmp()` pushes the current thread-context to the stack and continues execution. `longjmp()` restores the thread context later and thus provides the out-of-context jump for exceptions. Checking the value of the program counter on the stack would lead to a Time-of-check-Time-of-use problem (TOCTOU) [16]. Because the code in the library function performs the actual jump, our approach cannot be used without changing the implementation of `longjmp`.

Signals and Interrupts.

Signals and interrupts provide a mechanism for Inter-Process Communication (IPC) [32]. If a signal or interrupt occurs, a so called handler function is called. Because new handler functions can be registered at runtime, an attacker could in principle alter these handler addresses and redirect the control-flow to a point of her choice. The OS is responsible for determining and executing the handler function. Thus the jump is not executed in the checked program.

However, using signals and interrupts is deprecated on iOS and not supported by Objective-C.

Inline assembly.

For purposes of optimization and broadening the language semantics, high-level languages sometimes allow embedding assembler code. For example, the `setjmp-/longjmp`-mechanism is implemented in this fashion, because the C-programming language does not directly support this kind of control-flow transfer. We do not support it for two reasons: First, it would effectively mean to support a second programming language. Second, inline assembly can violate the semantics of the high-level language (e.g., by breaking the stack frames).

3. GENERAL APPROACH

There are many ways to mitigate certain types of control-flow hijacking attacks, but *Control-Flow Integrity* (CFI) [1] counters them thoroughly. The idea is to make sure that each instruction transfers the control-flow only to certain targets. In practice, this means that only jumps to "allowed" destinations are executed. While the root cause of a control flow hijacking attack (i.e., the write to a jump destination) is not prevented, it is ensured that the resulting control-flow cannot be arbitrarily chosen by the adversary.

One could reach this with means of correctness proofs of software [35]. Safe programming languages [44], where the semantics of the languages guarantee certain properties like checked array-boundaries, are as well possible as proofs on the values of pointers. These approaches are rather impractical, because proofs tend to be complicated and software is unlikely to be completely rewritten in a new programming language for security reasons only. The other possibility is to enforce those properties at runtime, by inserting code which checks them and aborts on violation [21]. While this means a slowdown of the program, it is not affected by a mismatch between high-level semantics and low-level behavior.

As previously mentioned, there are some problems with CFI tools based on binary analysis, especially for iOS. Instead, we propose to use a compiler-based approach. This means analyzing the source code, respectively a intermediate representation (IR), to extract the information about allowed and forbidden jump targets. Thus, we can avoid the problems related to app encryption and code signing. Also, we do not need to reverse engineer the program, but can make use of the semantic information provided by the programming language. Also, one can implant the constraints on the control-flow during the compilation phase. This allows to avoid trampolines and additional effort at load time due to problems with signed code or binary rewriting. In this paper, we use a compiler-based approach to insert runtime checks on the target of every jump before the jump is executed. Thus, we enforce CFI to counter control-flow hijacking attacks.

3.1 Notions

(Observed) Control-Flow.

We receive what we call the *(observed) control-flow* of the program by logging the state of the program counter with the execution time and the executing thread at runtime. This list is then sorted by thread and time. A single row of this list resembles the execution of the instruction the program counter points to in a certain thread at a certain time. While the exact resolution of the time and the thread ID are not important for our purposes, we demand that no two threads have the same ID and that no two executions in a single thread occur at the same time. Thus, we assume an exact ordering of the executions.

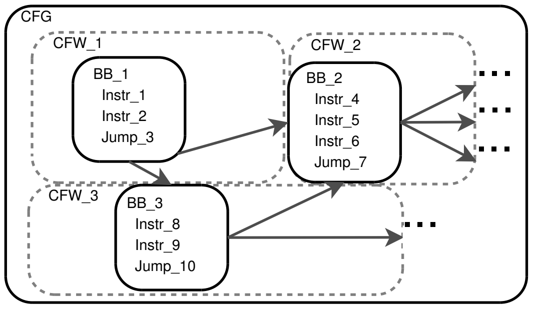

Figure 1: Relation between CFW and CFG

Normal instructions vs. jumps, Basic Block (BB).

We denote *normal instructions* as instructions that always pass control to the immediately following instruction. Thus, their effect on the program counter is merely the addition of their own size. Every instruction that does not behave like this is considered a jump. This applies to constants, conditional and computed jumps, function calls and returns, etc. It also refers to conditional jumps whose condition does not trigger: The instruction may, but must not, do something else to the program counter than adding its own length. Further, we demand that the program is split into basic blocks (lists of normal instructions, terminated by a jump).

Control-Flow Wish (CFW).

Each instruction has a list of "allowed" successors. By "allowed" we mean that the semantic of the programming language and the intent of the programmer do not object to transferring the control-flow from that instruction to the successor. For normal instructions, the successor is always the next instruction. Jumps with a constant target, whose address is not taken from data memory, can also be ignored. A jump and its successors can be uniquely identified by their basic blocks. We denote a basic block together with a list of its "allowed" successors a *Control-Flow Wish* (CFW)

Control-Flow Graph (CFG).

We define the CFG of a program through its CFWs. The nodes of this directed graph are given by the program's basic blocks. We draw an edge from one node to another, if the source nodes' CFW has the target node in its list of allowed successors. Thus, a CFW can be seen as a node of the CFG with its outgoing edges, as illustrated in Figure 1.

(Modeled) Control-Flow.

We can model the same list structure we previously denoted a (observed) control-flow with a CFG. First we choose a set of threads and an image base of the program's executable. Then we follow a path through the CFG for each of these threads, picking a steadily rising time stamp of each instruction of each basic block determined by the path. If a control flow is determined this way, we denote it a *(modeled) control-flow*.

Control-Flow Integrity (CFI), CFI enforcement.

With the notation of a CFG, modeled and observed control-flows, we can give a definition of CFI:

> If there exists a modeled control-flow for the CFG of a program, which is equal to the program's observed control-flow, the program's CFI held.

We call measures *CFI enforcement* if they make sure that no attacker can violate a program's CFI.

Control-Flow Constraint (CFC).

We denote a small piece of code before each jump, which ensures that this jump can only jump to successors according to its CFW, a *Control-Flow Constraint* (CFC). Note that a CFC depends only on the CFW for this jump, not the whole CFG. To enforce CFI with CFCs in our attack model, a few properties must hold:

- Safe program initialization: Each thread is initialized with its program counter addressing the first instruction of a basic block on the CFG.
- No data memory dependency: To prevent TOCTOU problems and manipulable CFWs, CFCs must not use values from data memory.
  - The jump does not take its target immediately from data memory, but loads it in a register first.
  - The CFC checks that register against constant data from code memory.
  - All targets are in code memory at load time.
- No new manipulable jumps: The CFC is allowed to introduce new jumps for its conditional behavior, but those jumps must be constant.

The attacker can no longer execute arbitrary code, not even in the style of *return-oriented programming* (ROP) [39], but can only execute full basic blocks of the program in a restricted order. She is forced to take a path through the CFG, but might be able to choose the path. For example, she could alter authentication tokens in memory [12] to execute restricted code on the CFG. The very pessimistic choice of the attacker's abilities is responsible for this.

Note that we consider a static CFG, which allows for static CFWs and ultimately static CFCs. While notions for dynamic CFGs are possible (e.g., by a dependency on the execution history of the program), every CFI enforcement for dynamic CFGs would need a runtime component, which holds and updates the CFG in a way the attacker cannot compromise. Thus, they need support of Write Integrity Testing (WIT) [2] or the OS. A problem with static CFGs arises in the context of function calls. The `return` of a function must be able to return to each call site where its function is called. But at runtime, only one location, namely the one that actually did the call, is a correct return target. This concept cannot be grasped with static checks. One could solve this issue by cloning the `return`s function for each of its call sites [38]. This not possible for recursive functions and leads to a potentially exponential size overhead. Here, we decide to live with the imprecision and argue that the attacker still cannot leave the nodes of the dynamic CFG and still takes a path through the static CFG.

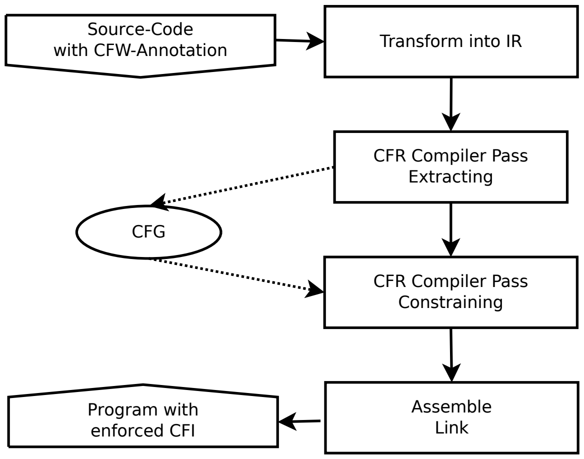

Figure 2: High-level overview of CFR's phases

Naturally, one can only constraint the control-flow of the processed code. Non-processed code occurs, e.g., in the context of plugins, JIT code, but also in the ambiguity of the Objective-C programming language and external libraries. For example, a library function has to be able to return to the locations it gets called from, but those locations are different for each program utilizing the library function. Additional effort of the OS or a runtime component could make this possible, e.g., by dynamically customizing the libraries CFG with the information from the program's CFG.

4. IMPLEMENTATION

We built our compiler-based CFI enforcement tool for iOS, called CONTROL-FLOW RESTRICTOR (CFR), using Clang [15] and LLVM [33], an extensible compiler framework. LLVM, which is also used in the Apple IDE XCode, has a low-level intermediate representation in SSA form and operates on it in so called passes. Those passes can analyze or transform modules or smaller constructs like functions or basic blocks. CFR was designed with the premise not to use a special runtime environment to eliminate the necessity for a jailbreak or a modified OS. It is implemented without modifying Clang or LLVM. We developed several LLVM passes in C++ that supply the necessary functionality. Those passes are described in the following, along with the necessary steps to process a program for CFI with our tool. Figure 2 provides a high-level overview of these steps.

Transform to LLVM-IR.

The first step merely consists of an invocation of Clang with the options for the iOS SDK, optimization, debug information, ARM code, and the emission of LLVM code. The debug information is necessary for CFR, but can be removed by LLVM later. We limit ourselves to a single instruction set at this point to simplify our task. We chose ARM code instead of Thumb code due to its richer instruction set.

Digression – Optimization.

The optimizer's assumption to be able to read data back unmodified after writing it into memory does not hold in presence of our attacker. Optimizations like register spilling or the partial removal of CFCs due to seemingly unnecessary checks on seemingly constant function pointers could destroy our scheme. Optimizations and their permutations are complex on their own, without the need to judge or track their effect on CFCs. We chose to carry out all optimizations before instrumenting the code, instead of picking a subset of non-destructive optimizations. Later optimizations on assembly level might also violate certain properties of CFCs. Verifying these properties in the IR is much simpler than doing so on assembler code. Therefore, we have disabled assembler optimizations for now.

Optionally, the programmer shall be able to explicitly influence the building of the CFG through annotations in the source code. These annotations are placed in source code comments, because we want to leave the source language and thereby the compiler unmodified. To match annotations to referenced jumps and targets, there has to be a certain parallelism between source code and IR. Optimizations might blur this parallelism, which is why we have to rely on debug-information at this point. Note that LLVM allows full optimization in spite of debug information as well as the removal of debug information at a later point.

Merging modules.

When two modules are analyzed to form a CFG, they cannot be processed individually. If one module calls a function in another module, the `return`-instructions in that function must be able to return to the calling modules' call site. Fortunately, LLVM provides a command-line interface to merge multiple LLVM-IR files into a single one. While this is less ideal in the context of a parallel or incremental compilation of multiple modules, it greatly simplifies our efforts.

Extracting semantic Control-Flow Wishes.

Our first implemented pass is invoked at this point. Its task is to extract the semantic CFWs from the supplied LLVM-IR. It does so by walking over the module, finding all function declarations (as they are possible destinations for function calls), and walking over all instructions of all found function definitions.

First, we determine the destinations for each jump. LLVM allows to distinguish constant jumps from indirect jumps, whose allowed destinations are also given. It is also possible to distinguish `objc_MsgSend`-calls from direct calls and indirect calls, whose type of function pointer can be inspected further. For the former, we have to retrieve its selector. If it is given at compile time, one can infer its indirectly accessed global variable from the call parameters.

In this process, every jump or jump destination is marked such that they can be mapped to the CFWs. Forbidden constructs, like inline assembly and the like, are also found and warned about in this phase. After all jumps and their possible destinations are retrieved, they are processed into CFWs. Each CFW holds various information, like a unique ID and the source code location of the jump. It also holds a tag on where the CFW was defined: *semantic*, for a CFW, which results from the semantics of the programming language, *programmer*, or *override* (see next paragraph for the latter two). Most important, the CFW holds a list of "allowed" and "forbidden" destinations. We chose the XML-format, supported by the pugixml framework [27], to store a CFW and to make it extensible and inspectable.

Extracting the programmer's Control-Flow Wishes.

We allow optional annotations in source code comments to influence the CFG. Through debug information and semantic CFWs, the annotations are set into context. This allows convenient referencing of functions, labels or call sites, without giving information such as line numbers. Only further restrictions of the semantic flow graph are allowed. These programmer-CFWs are supplied with the *programmer*-tag. Per default, the CFG allows all semantically sound jumps. This massively restricts an attacker and makes annotations necessary only for the dynamic behaviour of Objective-C, an even tighter CFG or for small performance improvements. Annotations might be problematic: The programmer might not allow a necessary jump, which would lead to a program crash. Since the ID of the jump and the faulty address are given in the error message, he can easily correct that mistake. Contrary, he might allow an otherwise unnecessary jump. While this might lead to security issues, we stress that possible targets are very restricted: One cannot jump in between instructions, and has to target the start of a function, a label or, in case of a return, a call site.

However, there is also the possibility to extend the CFG by explicitly using the CFW keyword *override*. This is necessary for certain dynamic behaviour of Objective-C. For example, the `forwardInvocation:`-method, which gets triggered for non-existent message selectors, must be allowed to return to its call sites. For selectors generated at runtime and called with a `performSelector:` Objective-C call, the programmer has to annotate this call site. Of course, the programmer's effort to annotate jumps might be assisted by further static or, even better, dynamic analysis techniques.

return in main().

A `return` in the `main()`-function returns to the operating system. Since the point from which the operating system calls it is not determined at compile time, we cannot check against it. Provided that no other function in the program calls `main()`, we can apply a simple fix: Instead of using the `return`-instruction in the `main()`-function, we use an `exit()` system call.

Inserting Control-Flow Constraints.

Now we insert a CFC before each non-constant jump. Their functionality is simple and straightforward: For each allowed jump destination according to the CFW, it shall execute the jump, for all others it shall abort with an error

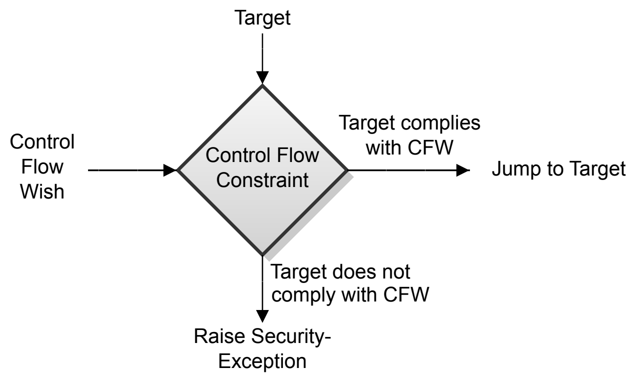

Figure 3: Functionality of a CFC

message (see Figure 3). The error message consists of the information that there was most likely a security violation, the ID of the jump, and the bad jump address. While the first serves as an information for the user, the latter can help the programmer to fix the issue.

The checks of the CFC are implemented in three assembler instructions: The first loads an allowed destination from code memory into a register. The second compares this register to the register that holds the target address. On equality, the third instruction jumps over the other checks and the call of the error-and-abort function, to the actual jump. On inequality, there is either another check or the error-and-abort function is executed. Listing 1 shows that a CFC is essentially a linear check against the whitelist of allowed destinations.

```
   if(target == allowed_1) goto JUMP;
     ...
   if(target == allowed_n) goto JUMP;
  security_exception();
JUMP:
 Jump to target;
```

Listing 1: Pseudo-Code of a CFC

Note that `objc_MsgSend`-calls require additional care: We apply the same principle to check against selectors, but also have to check the value of the casted pointer to the API function. Also, we can assume the presence of the jump destinations for all other jumps, but not for selectors, which were supplied by programmer CFWs. Thus, we may have to insert a selector into the global variables.

Assembly.

We operated on the LLVM-IR so far and need to translate it to assembler code now. The necessary command-line tool is provided by LLVM. We ensure that the executable is not compiled as position independent code, as this implies the use of relative addresses, which would complicate inserting the checks. For that reason ASLR [40], which conventionally helps protecting the control-flow, cannot be used, except in shared libraries. ASLR also hinders attacks on data, but since neither the principle of our approach nor its runtime behaviour change, we argue that our prototype can do without ASLR.

Relocating CFCs.

The jumps in the LLVM-IR are sometimes not represented with a single assembler instruction. For example, a `return` is not only the actual jump, but also restores the stack frame. Thus, the CFC is not always directly in front of the jump. To rectify this, we can simply move it forward, while taking care not to overwrite needed registers.

5. EVALUATION

Based on our current implementation, we now discuss the properties of our compiler-based CFI enforcement tool CFR. We provide analytical information of code size and runtime of the CFCs, show the broad applicability and demonstrate its efficiency by applying it to programs. For benchmarking, we used an iPod Touch 4G with iOS v4.3.3, 256 MB DRAM and an Apple-A4 (Cortex A8) CPU @ 800 MHz.

5.1 Security Reasoning

Observed and modeled control-flow basically capsule the idea that for CFI to hold, the attacker must not be able to leave the CFG. She cannot directly set the *program counter*, therefore she must utilize the instructions of the app to alter the control-flow. When we assume that the program is initialized to start its execution on the CFG, two kinds of instructions can follow. First, there are non-jump instructions. They cannot leave the CFG, or even their basic block, by definition. Second, there are jumps, which are constrained by our approach through a CFC. By construction, the target is held in a register and compared against a whitelist from code memory. Because each entry in the whitelist points to the CFG, and because the attacker cannot alter the whitelist or the jump destination before a jump, this prohibits her from leaving the CFG. Therefore, our tool enforces CFI under the given assumptions.

5.2 Analytical Properties of CFCs

A CFC consists of a series of checks and a call to the abort function. The latter consists of three instructions to load the jump ID, to load the non-allowed jump destination, and the actual call to the abort-function. Additionally, the 32-bit jump ID has to be saved, which makes a total of 16 bytes.

There is one check for each allowed destination of the specific jump. Each check consists of three instructions as well: One to load an allowed destination, one to compare it to the would-be-used destination, and a jump for the case of equality over the abort function to the actual jump. Again, with the 32-bit value for the allowed destination, we reach 16 bytes. For `objc_MsgSend`-calls, each CFC has an additional check to verify the pointer to the API function. For calls using `performSelector:` or `NSInvocation.invoke`-like methods, an additional check for their selectors is included.

The inserted code adds to runtime by the execution of the checks, as the time to load and handle the increased code size is rather negligible. Without an attack, a CFC executes a number of non-equality detecting checks, followed by a single equality detecting check. Because the second kind executes a jump, while the first one does not, its runtime is a little higher. In a micro benchmark, we measured about 9.898ns on average for a single equality detecting check, while an non-equality detecting check takes about 5.449ns.

5.3 Benchmarking

Real-world apps usually perform multiple smaller chunks of computations, triggered by various events like user input, network traffic or file I/O. These events have their own individual delay, which in sum usually dominate the runtime of the app and are not influeced by our instrumentation. Therefore, runtime overhead due to CFI can hardly be measured objectively in real-world apps.
However, we wrote a simple timing app that allows embedding of different sub-programs for our performance evaluation. Our benchmarking programs are deterministic and work on a single source of random input data. The time to load their input from disk to memory is not part of the measurement, which means that each test program does operate solely on RAM without any further I/O. Together with the fact that they where chosen to perform a lot of guarded jumps compared to normal instructions, their runtime overhead can be seen as an upper-bound for real-world apps.

Program	*Inter-preter*	*Game of Life*	*AES 256*	*RC4 256*	*Quick-Sort*
Functions	7	11	17	4	5
NCLOCS	147	159	303	70	60
Asm-Instructions	604	564	1034	185	224
Returns	29	22	63	4	5
ø Checks/Return	3.22	2.37	4.2	1	2
Indirect Branches	7	0	0	0	0
ø Checks per indir. Branch	0	7	0	0	0
Indirect Calls	0	1	0	0	0
ø Checks per indir. Call	0	3	0	0	0

Table 1: Properties of test-programs (-O0)

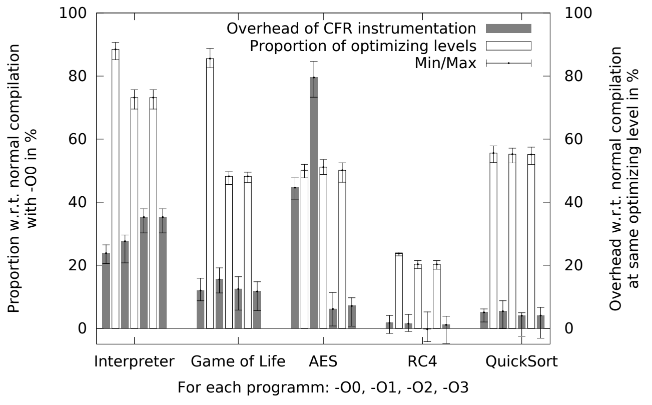

Figure 5: Measured runtime overhead

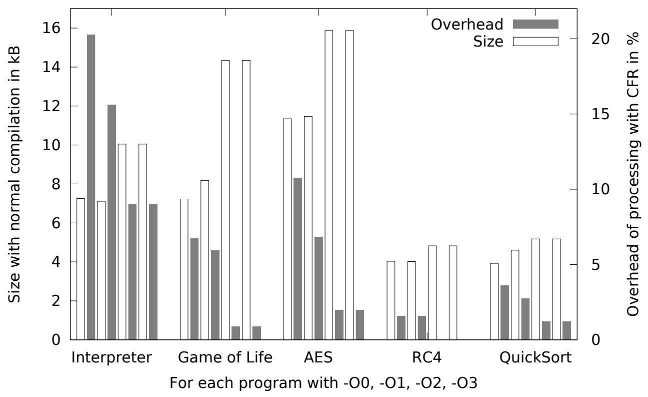

Figure 4: Size overhead of the binary

Table 1 provides an overview of the different programs and we briefly review each of them

Our first benchmarking program is a Game of Life [23] implementation with an interchangeable survival rule, which is called via an indirect function call. The game consists of a two-dimensional array of cells. The aliveness of a cell is determined by the survival function, which has the surrounding cells as input. Thus, this function is called for every cell in each transition of the automaton. The second program is an interpreter for a simple programming language. It processes a series of opcodes, which encode single instructions. Using the current opcode as input for an indirect jump, it jumps to the handler for the next opcode. Naturally, this is done for each instruction and since any opcode might follow, the number of potential jump-destinations is high for each jump. We also processed two well-known crypto algorithms. First, we used an unoptimized, modular AES implementation [31]. Since the low-level functions get called very often, it executes many `return` instructions. The second crypto algorithm is RC4 [4]. One example of recursive algorithms is QuickSort [26]. Since we argued about the imprecision in the return destinations, even in the presence of cloning, this is a nice example. Calls and returns are so frequently executed in this algorithm that it served as a worst-case example for MoCFI [20].

Figure 4 shows that the binary gets bigger for higher optimization levels, which is mainly due to function inlining. Because of this, there are less `return`s in the program. This reduces the number of CFCs, which reduces the relative size overhead of using CFR. We observed an overhead of less then 10% for the tested programs at the standard optimization level `-O2` [24], where especially the number of returns with multiple checks seems to have a high impact. One has to keep in mind that the ratio between jumps and normal instructions is high in these programs. Thus, real-world apps are likely to have a lower relative size overhead. For RC4, the optimizer completely inlines the functions, leaves no `return` left and removes the need for a CFC. Further, we observed no difference between the `-O2` and `-O3` optimizations, which would be different for more complex programs.

We chose the input-size for each program such that the execution takes roughly ten seconds for the uninstrumented, unoptimized code (`-O0`). As expected, Figure 5 shows that the runtime decreased for higher optimization levels. For each tested program, the runtime overhead was below 35% at the standard optimization level `-O2`. This value was reached for the especially disadvantageous case of the simple interpreter: its operations (`push, pop, inc, dec` and the like) are very simple. Thus, one has many jumps with many allowed targets for relatively few instructions. For the other programs, we stayed even below a 15% runtime overhead. Even this value has to be seen in the context of these programs. They work completely on RAM and do not wait for network latency, disk access time or user input. For real-world apps, the runtime overhead would most likely be hardly noticeable for the user. The lack of CFCs in RC4 due to the extreme compiler optimization can also be seen in the runtime overhead.

While the type of jump is rather irrelevant, its number of checks has an impact. This will usually mainly affect `return` instructions, especially in modular implementations. Arguably, a high fan-out only matters if its jump is executed frequently. For large functions, the ratio between non-jump instructions and jump instructions will be high enough, such that the runtime is not dominated by the jump instructions. For small functions, this would not be the case. However, the compiler will most likely inline such functions. The sudden drop in runtime for higher optimization levels in the AES example illustrates this.

5.4 Broad Applicability

As stated above, it is complex to measure the runtime overhead in real-world apps. Also, apps in the AppStore are also usually not open-source. While the authors of the apps could apply our tool on them, we simply cannot do it. However, we feel it is necessary to show that CFR can compile complex apps, while preserving their normal ability to run. Thus, we compiled and ran apps provided as sample code on Apple's developer website [10]. While these ∼150 apps are simple, they use a widespread spectrum of possible capabilities of an iOS app. Thus, we believe our tool to be broadly applicable.

6. RELATED WORK

Due to the potential impact of a control-flow attack, both attack strategies and countermeasures are widely studied. A program's vulnerability against such attacks may be introduced by programming mistakes, e.g., by using vulnerable functions (e.g., `strcpy()`) or flawed array boundary checks. Thus, the first countermeasure is to find and fix those bugs (e.g., [45] or [25]). Further, one can design a language such that certain bugs cannot appear (e.g. [44]) or one can attempt to prove their absence (e.g. [35]).

Many techniques hinder an attacker in the way she puts her attack on control-flow in execution. For example, the stack is made non-executable [41], one can use stack canaries [19], or libraries are loaded to different addresses [40]. Several other defense strategies exist as well [18].

Starting with *program shepherding* [29], many techniques were proposed to prevent runtime attacks. One can prevent many attacks on the control-flow by protecting the write-integrity of control data. The resulting technique is called *Write Integrity Testing* (WIT) [2]. There are multiple different approaches to reach CFI. NaCl [46] sandboxes external code for browsers. They use alignment to reach a coarse CFI, only limiting the attacker to target complete instructions. Abadi et al. [1] operate solely on the binary of an x86-executable for Windows before load time to extract the CFG and to insert checks. They place an ID before possible destinations. Each instrumented jump then checks that ID against his own. This seems less feasible for compiler-based approaches, especially with external code, since each ID must be unique in the binary code of the whole program. However, Strato [47], a recent compiler-based framework for inlined reference monitors (IRM), seems to have solved this problem for x86 Linux. Zhang and Sekar [49] implement very efficient CFI for x86 Linux based on binary analysis. They use trampolines, address tables and address translation. HyperSafe [42] protects Type-1 hypervisors without an underlying OS. In their approach, target addresses are used indirectly, but through their index in a static target address table. Thus, they can simply check, if the index points into that array. Problematic is that each assignment of a function pointer has to use the index instead, which is impossible for dynamic code or calculated jumps. Also, the stack-frame layout is changed. These approaches use LLVM, are for x86-platforms and need to modify the source code.

MoCFI [20] operates on iOS/ARM-binaries to extract the CFG. Since there is no sufficiently mature binary-rewriting engine for iOS/ARM, they have to use trampolines that redirect to a runtime component instead of static checks. Because iOS-binaries are code-signed, they have to perform hot-patching after load time to insert their checks. This requires a jailbroken device, a severe limitation in practice that we overcome with CFR.

WIT and CFI are not completely orthogonal. WIT relies on weak CFI properties to make sure that its checks are executed, while CFI relies on the write-integrity of certain constant data. XFI [22] ensures certain properties from both concepts in an executable. It is implemented for x86-Windows and uses binary rewriting.

7. LIMITATIONS

Our CFCs are essentially linear checks against whitelists of allowed target addresses. Thus, jumps with many possible targets are slower than those with fewer targets. While heuristic ordering or a tree-based data structure might help, other schemes provide constant runtime per CFC [1, 42]

While we support the most common jumps types, several types cannot be supported easily. Especially exceptions are an open and pressing problem for CFI on iOS. The `setjmp()/longjmp()`-mechanism prohibits instrumentation in the app itself. However, under the term Zero-Cost Exception (Itanium-ABI and 64-Bit Mac OS X) [6] and SafeSEH (Windows) [34] another mechanism is used, which stores the catch locations in a constant table. This table is not manipulable, which prevents the attacker to alter addresses of exception handlers. Thus, a CFI enforcement tool could rely on the OS to support exceptions without further effort.

Libraries, plugins or the like need customization. Currently, without a runtime environment or OS modifications to accumulate each modules CFWs at load time, such code has to be recompiled for each program.

One also might think about porting our tool to other languages, processors, or operating systems. While we basically support the full C-semantic (apart from the return-imprecision), which is common ground for many imperative languages, we make certain assumption on the OS (like the $W \oplus X$-policy or no code-loading at runtime). The assumptions on the processor are largely restricted to the implementation of the checks and the verification phase.

8. CONCLUSION

We have presented CFR, a compiler-based approach to enforce control-flow integrity on iOS apps. Compared to previous work in this area, our approach can be deployed without requiring a jailbreak of a given device. Furthermore, by directly integrating CFI enforcement checks during the compilation phase, the performance impact is reduced to only three instructions to instrument a given control-flow decision. We have implemented a prototype of our approach and directly integrated it into a compiler. Empirical measurement results on an implemented prototype indicate the practical viability and low overhead of this approach.

Acknowledgements

This work was supported by the DFG as part of the research training group *UbiCrypt* (GRK 1817).

9. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

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

[3] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7, Nov. 1996.

[4] Anonymous. Posting of RC4-Algorithm, Sept. 1994.

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

[6] Apple. Exceptions in 64-Bit Executables, Feb. 2010.

[7] Apple. App Store Review Guidelines, Apr. 2012.

[8] Apple. iOS App Developing Guide, 2012.

[9] Apple. The Objective-C Progamming Language, 2012.

[10] Apple. Sample code for apps, Feb. 2013.

[11] ARM Ltd. The ARM Processsor Architecure, 2012.

[12] A. Baliga, P. Kamat, and L. Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *IEEE Symposium on Security and Privacy*, 2007.

[13] P. Bellmer. Market share of Android, iOS and Bada rizes, May 2012.

[14] Blexim. Basic Integer Overflows. *Phrack*, 60, 2002.

[15] Clang Project. Clang - A C Language Family Frontend for LLVM, May 2012. v3.0.

[16] Common Weakness Enumeration. CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition.

[17] Common Weakness Enumeration. CWE-134: Uncontrolled Format String, May 2012.

[18] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard^{TM}: Protecting Pointers From Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*, 2003.

[19] C. Cowan, C. Pu, D. Maier, H. Hinton, 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.

[20] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[21] Erlingsson and Schneider. IRM Enforcement of Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, 2000.

[22] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[23] M. Gardner. Mathematical games - the fantastic combinations of john conwayś new solitaire game life, Oct. 1970.

[24] GCC. GCC Manual - Chapter 3.10, 2013.

[25] E. Haugh. Testing C programs for buffer overflow vulnerabilities. In *Symposium on Network and Distributed System Security (NDSS)*, 2003.

[26] Hoare. Quicksort. *The Computer Journal*, 5, 1962.

[27] A. Kapoulkine. pugixml: Light-weight, simple and fast XML parser for C++ with XPath support, May 2012.

[28] M. Keller. Geek 101: What is jailbreaking?, Feb. 2012.

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

[30] C. Lattner. Address of Label and Indirect Branches in LLVM IR, Jan. 2010.

[31] I. O. Levin. A Byte-oriented AES-256 implementation, 2009.

[32] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram, and U. Drepper. The GNU C Library, Chapter 24: Signal Handling, Aug. 2001. v0.10.

[33] Low-Level Virtual Machine Project. , Nov. 2011. v3.0.

[34] Microsoft. The safeSEH-switch for Safe Exception Handling, 2005.

[35] G. C. Necula. Proof-carrying Code. In *ACM Symp. on Principles of Programming Languages*, 1997.

[36] F. Nidito. Exceptions in C with Longjmp and Setjmp, 2012.

[37] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. In *IEEE Symposium on Security and Privacy*, July 2004.

[38] J. V. R Gopalakrishna, E Spafford. Efficient intrusion detection using automaton inlining. In *IEEE Symposium on Security and Privacy*, 2005.

[39] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1), 2012.

[40] 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.

[41] Solar Designer. Non-executable stack patch.

[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] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Symposium on Network and Distributed System Security (NDSS)*, 2003.

[44] C. Wong. Buffer Overflow in Java, 2012.

[45] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[46] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*, 2009.

[47] B. Zeng, G. Tan, and U. Erlingsson. Strato - A Retargetable Framework for Low-Level Inlined-Reference Monitors. In *USENIX Security Symposium*, 2013.

[48] 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*, 2011.

[49] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*, 2013.