Subversive-C: Abusing and Protecting Dynamic Message Dispatch

Julian Lettner, *University of California, Irvine;* Benjamin Kollenda, *Ruhr-Universität Bochum;* Andrei Homescu, *Immunant, Inc.;* Per Larsen, *University of California, Irvine, and Immunant, Inc.;* Felix Schuster, *Microsoft Research;* Lucas Davi and Ahmad-Reza Sadeghi, *Technische Universität Darmstadt;* Thorsten Holz, *Ruhr-Universität Bochum;* Michael Franz, University of California, Irvine

https://www.usenix.org/conference/atc16/technical-sessions/presentation/lettner

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16) is sponsored by USENIX.

Subversive-C: Abusing and Protecting Dynamic Message Dispatch

Julian Lettner[*] *Benjamin Kollenda*[†] *Andrei Homescu*[§] *Per Larsen*[*§] *Felix Schuster*[¶]
Lucas Davi[‡] *Ahmad-Reza Sadeghi*[‡] *Thorsten Holz*[†] *Michael Franz*[*]

[*]*UC Irvine* [†]*Ruhr-Universität Bochum*
[§]*Immunant, Inc.* [¶]*Microsoft Research*
[‡]*Technische Universität Darmstadt*

Abstract

The lower layers in the modern computing infrastructure are written in languages threatened by exploitation of memory management errors. Recently deployed exploit mitigations such as control-flow integrity (CFI) can prevent traditional return-oriented programming (ROP) exploits but are much less effective against newer techniques such as Counterfeit Object-Oriented Programming (COOP) that execute a chain of C++ virtual methods. Since these methods are valid control-flow targets, COOP attacks are hard to distinguish from benign computations. Code randomization is likewise ineffective against COOP. Until now, however, COOP attacks have been limited to vulnerable C++ applications which makes it unclear whether COOP is as general and portable a threat as ROP.

This paper demonstrates the first COOP-style exploit for Objective-C, the predominant programming language on Apple's OS X and iOS platforms. We also retrofit the Objective-C runtime with the first practical and efficient defense against our novel attack. Our defense is able to protect complex, real-world software such as iTunes without recompilation. Our performance experiments show that the overhead of our defense is low in practice.

1 Introduction

The primary programming environment on Apple's OS X and iOS platforms uses a language called Objective-C, which extends the C language with object-oriented constructs. Many of the main application programs on Apple's platforms, such as Safari, iTunes, etc. are built using Objective-C, which differs from C++ in the way that dynamic dispatch of function calls is implemented. In spite of its importance to commercial software platforms, it has attracted little scrutiny from systems security researchers.

The latest code-reuse mitigation being deployed—CFI—makes traditional ROP [30] attacks harder to construct. CFI computes an approximation of an application's control-flow graph (CFG) and verifies that all indirect branches follow valid CFG edges at run time [1]. In contrast to randomization-based defenses [26], CFI is secretless and cannot be bypassed via information leakage. Like other mitigations, CFI must trade off security (precision) for performance. Coarse-grained CFI policies [43, 44] leave a small fraction of code locations available for reuse by adversaries—enough to mount ROP attacks [16, 22, 32]. The deficiencies of coarse-grained CFI renewed interest in more precise policies. Devising such CFI policies typically requires source code access, because structural information required to compute a complete and precise CFG is lost during compilation. The recent COOP [31] code-reuse technique exploits the imprecision of non-C++ aware CFI implementations on Windows and Linux. Specifically, the attacker manipulates the *virtual method tables (vtables)* of C++ objects in memory such that a sequence of attacker-chosen regular virtual methods is executed via likewise regular virtual method call sites. Unlike ROP, COOP does not violate the integrity of return addresses or produce corrupted call stacks and therefore remains undetected by generic CFI policies [17, 20]. Moreover, the high-level structure of C++ code (e. g., class hierarchy and dynamic object types) cannot be fully recovered without source code, so malicious COOP control flows are difficult to distinguish from benign ones even for *C++-aware* CFI policies computed by binary analysis. In terms of expressiveness and flexibility, COOP is comparable to ROP in C++ environments [14, 31]. Still, it remains unclear whether COOP is limited to C++ code on Windows and Linux or whether it is a generic threat on par with ROP.

This paper shows that programs written in Objective-C suffer from a systematic vulnerability that enables COOP-style exploits against Objective-C on OS X and iOS. Like C++, Objective-C extends the C programming language with object-oriented constructs. Although both languages add *dynamic dispatch* of function calls to C, the implementation of this feature differs greatly between C++ and Objective-C. Whereas C++ fixes the vtable for each class at compile time, Objective-C enables full *late binding* by (re)mapping literal method names to actual functions dynamically at run time. We dub our new class of attacks Subversive-C and demonstrate its viability against applications using AppKit, a commonly used framework on Mac OS X, by constructing a proof-of-concept exploit.

We also show how Subversive-C exploits can be mitigated. Our mitigation strategy can be retrofitted onto existing systems without requiring recompilation of the programs being protected and has very little overhead.

An important insight is that in many cases, an attacker can use COOP, Subversive-C, or a combination of both, because non-trivial OS X and iOS applications like Safari or MS Office typically contain both Objective-C and C++ (standard libraries or ported code from other platforms) components. In fact, it is even valid (and common) to tightly interweave Objective-C and C++ semantics. Such "Objective-C++" code is accepted by the GCC and Clang compilers. Hence, effective code-reuse defenses for OS X and iOS need not only to consider high-level semantics of Objective-C, but also those of C++.

In summary, our main contributions are as follows:

- **Novel Offensive Technique** We present Subversive-C, a new offensive technique that reuses entire Objective-C methods by carefully arranging the metadata used to dispatch messages in the Objective-C runtime. The dynamic nature of Objective-C coupled with whole-function reuse renders existing integrity and randomization-based defenses ineffective against Subversive-C exploits.
- **Hardened Objective-C Runtime** Because existing defenses cannot protect against Subversive-C with low overheads, we developed a new defensive technique to prevent adversaries from manipulating and corrupting metadata used by the Objective-C runtime. Specifically, we retrofit the Objective-C runtime with integrity checks in the lookup processes that handle Objective-C message dispatch. Our hardened runtime is fully compatible with the runtime shipped with OS X and can protect complex, real-world applications such as iTunes.
- **Realistic and Extensive Evaluation** We demonstrate a fully-fledged Subversive-C attack targeting the AppKit library. We also provide a careful and detailed evaluation of our hardened Objective-C runtime. We report a 1.54 % aggregate overhead for complex, real-world applications.

2 Technical Background

In the following, we provide a brief overview of the technical concepts we use in the rest of this paper. We discuss dynamic message dispatch in Objective-C and present an overview of resarch on code-reuse attacks with a specific focus on COOP.

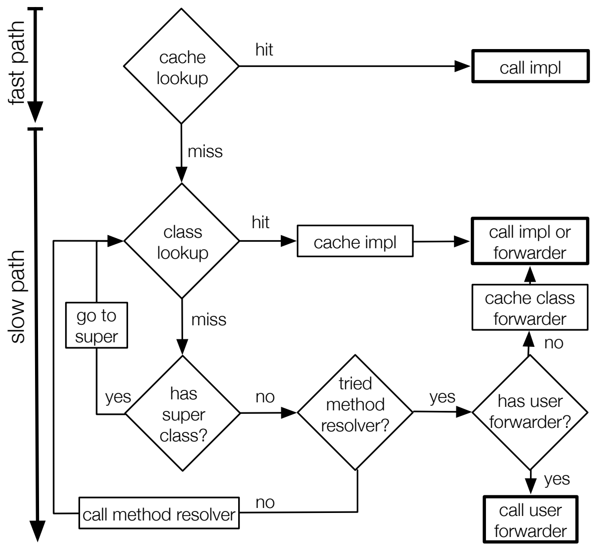

Figure 1: Fast and slow paths when dispatching messages.

2.1 Dynamic Message Dispatch

Objective-C is an object-oriented programming language that extends the C language with dynamically dispatched Smalltalk-style messaging. Where C++ programmers invoke (virtual) methods of objects, Objective-C programmers send messages to objects. Each message has three components: i) the **receiver** object; ii) the **selector**, an identifier naming the method that receives the message; and iii) zero or more **arguments**.

Although Objective-C is a statically compiled language, the targets of message dispatches are resolved at run time. At every message dispatch location, the compiler simply emits a call to the `msgSend` function (or one of its variants) in the Objective-C runtime. The purpose of the `msgSend` function is to locate the appropriate method for a given (receiver, selector) pair and subsequently execute it.

Figure 1 illustrates the message dispatch algorithm as implemented in Apple's Objective-C runtime. It consists of a fast path and a slow path. The slow path retrieves the method implementation corresponding to a given selector by searching through all methods defined by the class of the receiver object and all its ancestors. The search operates on compiler-generated metadata attached to each object as shown in Figure 2.

The lookup algorithm starts with the class of the receiver and checks the selector against all methods defined by the receiver's class. If no method is found, the methods of the parent class is searched until a method implementation is found or the root of the class hierarchy is reached. If neither the class itself nor any of its ancestors contain a method implementation, the runtime allows the class to

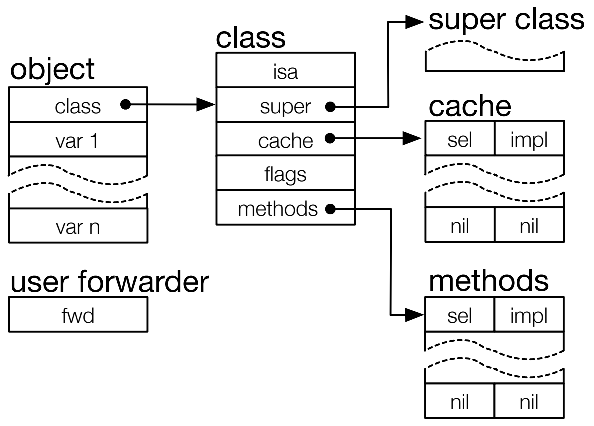

Figure 2: Layout of objects, classes, and lookup caches.

dynamically add an implementation for the given selector. If the class provides a "method resolver" function, the runtime calls it. The resolver (depending on its implementation) may add a new method to the class. The runtime then repeats the entire lookup process, in case the method added by the resolver corresponds to the input selector.

The fast path speeds up message dispatch by storing the results of each lookup in a per-class cache, and reuses previous results if available. At the beginning of each message dispatch, the runtime queries this cache for a method pointer as shown near the top of Figure 1.

The cache entries are stored in memory as a linear array of (selector, method pointer) pairs. The class metadata includes a pointer to its corresponding cache. The runtime performs the lookup using a linear probing algorithm. The lookup starts from a location computed by hashing the selector itself and proceeds linearly through the array until either a match is found—a *cache hit*—or an empty entry is reached—a *cache miss*.

In case neither the cache lookup nor the slow class hierarchy walk find the method for a selector, the runtime performs one last step before exiting with an error. If the class provides a "forwarder" function, the runtime calls this function with the selector, allowing classes to dynamically respond to new messages at run time or forward messages to other objects. Additionally, the Objective-C runtime allows the application to install a "user forwarder" that overrides all per-class forwarders. If this handler is installed, the runtime calls it at the end of the method lookup process. The handler pointer is stored in a writable global variable, which can be manipulated by adversaries.

2.2 Exploitation and Code-Reuse

C/C++/Objective-C eschew memory and type safety features of modern languages and require manual memory allocation and deallocation. This leads to a steady stream of memory management errors.[1] Attackers exploit the presence of these errors to craft malicious inputs that hijack the control flow of the application. The classic stack smashing attack injects code and redirects execution to it by overwriting the return address stored past the end of an overflowed buffer [2]. Thanks to modern mitigations such as data execution prevention (DEP), which disallows memory regions that are both writable and executable, code injection is all but obsolete. Therefore, modern exploits reuse legitimate code to bypass DEP. There are many known variants of code-reuse attacks. The main differences are the granularity at which legitimate code is reused. ROP reuses short instruction sequences ending in returns called gadgets [30,34]. Other variants reuse whole functions, including Return-into-`libc` (RILC), COOP, and our novel Subversive-C technique. Another key difference is the dispatching mechanism used to transfer control between the code snippets being reused. ROP and RILC use return instructions or indirect jump/call instructions [11]. COOP-style attacks use special "main-loop gadgets" to iteratively or recursively dispatch a sequence of method calls controlled by a malicious payload.

An important prerequisite of a code-reuse attack is knowledge of the target's memory layout, because the payload in a code-reuse attack necessarily (directly or indirectly) references existing code locations. Thus, address space layout randomization (ASLR) complicates code-reuse attacks because it randomizes the base address of linked libraries at load time. However, this is only a small hurdle for a practical code-reuse attack since information leakage or memory disclosure attacks often enable attackers to undermine ASLR [33,35,36,38].

2.3 Counterfeit Object-oriented Programming

COOP is a code-reuse technique targeting C++ software [31]. In COOP, a sequence of attacker-chosen C++ virtual methods (also called *vfgadgets*) is executed on attacker-injected objects (also called *counterfeit objects*). Each vfgadget in such a sequence fulfills a specific task, such as reading a value into a register and may have certain side-effects. Executed one after another, the vfgadgets implement the malicious functionality desired by the attacker, e. g., the execution of a shell command. Put simply, (short) virtual methods are to COOP attacks what gadgets are to ROP attacks. Whereas a ROP attack is initiated by injecting a "fake stack" (containing fake return addresses) into the target address space, a COOP attack injects a collection of "counterfeit objects", typically using a single attacker-controlled write. Each counterfeit

[1] For new code, disciplined use of modern coding techniques like smart pointers for C++ and automatic reference counting (ARC) for Objective-C alleviate this problem.

object corresponds to exactly one vfgadget and carries a corresponding pointer to a vtable.[2]

In ROP and related techniques, data primarily flows through registers and the stack from one gadget to another. In contrast, data may flow in three different ways between COOP vfgadgets: *(i)* through method arguments, *(ii)* through global variables, and *(iii)* through member fields of *overlapping* counterfeit objects. The latter is a pattern specific to COOP, which can greatly facilitate the creation of meaningful vfgadget chains. For example, vfgadget *1* may read a value from memory and store it in the field x of counterfeit object A and vfgadget *2* may increment field y of counterfeit object B. By making objects A and B overlap such that $A.x$ and $B.y$ map to the same address, the methods *1* and *2* can be used in conjunction to read and increment a value.

Different techniques for chaining the execution of vfgadgets in a COOP attack have been described in previous work [14, 31]. Using one of these techniques, the attacker initially corrupts a C++ object used by the target application such that a subsequent virtual method call is maliciously diverted to a central dispatcher vfgadget. (In a ROP attack, the control flow would instead be diverted to the first gadget, which usually pivots the stack pointer.) In the simplest case, a so called *main loop* (ML-G) vfgadget is used. Briefly, an ML-G is a virtual method that invokes virtual methods on a collection of C++ objects. By making an ML-G operate on a collection of counterfeit objects, the chained execution of vfgadgets becomes possible. An example ML-G is discussed in Section 4.2.

3 Threat Model and Assumptions

The assumptions underpinning this research are:

- **Data** The attacker can arbitrarily read and write data pages as allowed by the page permissions. Specifically, the internal data structures of the Objective-C runtime can be read, written, and corrupted. However, we assume that ASLR is in place to randomize the locations of program and runtime data structures.
- **Code** We assume that DEP prevents code injection by disallowing the execution of writable pages.
- **Runtime** We assume that the runtime is protected using fine-grained code randomization [26], as well as an implementation of execute-only memory (XoM), such as XnR [6], Readactor [13], or HideM [21], that prevents attackers from using information leaks to retrieve the code of the runtime. We also rely on these defenses to secure secret keys (see Section 5.4).
- **Control flow** Since Objective-C is a superset of C, we assume the C parts of the application and runtime are protected using appropriate mitigations (CFI, randomization, or equivalent defenses). Defenses such as Mobile CFI (MoCFI) [15] can be used to protect Objective-C code from control-flow hijacking.

Note that these assumptions are realistic and match the capabilities of a real-world attacker. They also match the adversarial models used in closely related work [13, 14].

4 Subversive-C

In this section, we demonstrate that the principles of the COOP attack are not only applicable to C++ but also to Objective-C. Conceptually, a Subversive-C attack proceeds analogously to a COOP attack: the attacker diverts an application's control flow such that a sequence of attacker-chosen Objective-C methods (vfgadgets) is executed on injected counterfeit objects. The first method executed in such a sequence is necessarily a dispatcher vfgadget, e. g., a *main loop vfgadget* (ML-G) as described in Section 2.3. COOP and Subversive-C are closely related in the way they rely on counterfeit objects and vfgadgets. However, as they target different programming languages, COOP and Subversive-C counterfeit objects differ in their layouts. For COOP it is sufficient to create objects that reference a vtable, whereas the Objective-C runtime features a more involved class layout. Therefore, an attacker must forge multiple data structures to launch a Subversive-C attack. The exact procedure is described next in Section 4.1. Section 4.2 then presents a concrete Subversive-C attack against applications that use the AppKit library.

For brevity, we limit the discussions in this section to Apple's OS X operating system and the x86-64 architecture. However, all techniques and concepts extend to Objective-C code running on iOS and ARM.

4.1 Exploiting the Objective-C Message Dispatch Mechanism

The Objective-C runtime implements two different ways (*slow* and *fast*, see Section 2.1) to resolve a class-selector pair to a function address. We now describe how the attacker can exploit the Objective-C runtime's slow path and fast path lookup mechanisms in order to control the methods invoked on counterfeit objects in a Subversive-C attack. These techniques are specific to Subversive-C and are the key differentiators with respect to COOP.

Slow Path As described in Section 2.1, when a cache lookup for a selector fails, the `msgSend` function does a slow search through all methods available for the receiver object. The corresponding data structures are partly stored in read-only memory and cannot be modified by the attacker at run time. Hence, in order to freely choose the

[2]In C++, every object of a class with virtual methods carries a pointer to the class's fixed vtable. Whenever a virtual method is to be executed on a C++ object at run time, this pointer is dereferenced and the respective method's address is fetched from the table.

vfgadgets executed in a Subversive-C attack, the attacker needs to inject new fake data structures alongside each counterfeit object. Concretely, each counterfeit object needs to reference its own fake *class struct*[3] which in turn references its own fake *method list* (cf. Figure 2).

Each entry in a class's method list links a function pointer to a selector. It is thus sufficient to inject fake method lists with a single entry. This entry must link the fixed selector used in the dispatcher gadget to the vfgadget that is to be executed on the corresponding counterfeit object. In turn, the injected fake class struct must be shaped in such a way that `msgSend` actually takes the slow path and evaluates the given method list as desired. A straightforward way to ensure this is to null-out the cache-related fields in the class struct (i.e., invalidate the cache) and to mark the class as *initialized* by setting the corresponding bit in the `flags` field (not shown in Figure 2).

Instead of creating valid class structs from scratch, for increased stealthiness and simplicity, an existing class struct that is compatible with the given dispatcher can be copied and modified as needed.

Fast Path Instead of invalidating the cache of counterfeit objects, the attacker can also opt to exploit the fast path look-up by injecting fake class structs with *valid* cache entries linking the dispatcher's selector to vfgadgets. Doing so is simple, as the caching mechanism does not use a secure hashing function and, in any case, its parameters can also be directly rewritten by the attacker. Hence, the attacker can arbitrarily precompute valid cache entries offline and incorporate them into fake class structs.

Forward Handlers In addition to forging method lists and caches, a third option for the attacker to execute arbitrary methods from a dispatcher is to abuse *forwarders*, which are introduced in Section 2.1: existing forwarders structs (cf. Figure 2) could be manipulated or fake ones could be injected such that vfgadgets are executed instead of actual forwarder handlers. In this approach, the attacker needs to make sure that both the slow and the fast path fail for all counterfeit objects for the given dispatcher—e.g., by injecting fake class structs with an invalid cache and an empty method list.

4.2 Proof-of-Concept Exploit

To demonstrate the general applicability of our technique, we constructed a Subversive-C attack for the x86-64 version of the AppKit library. AppKit is part of the Cocoa framework which encompasses Foundation, AppKit and Core Data. AppKit in particular is used to create graphical user interfaces. As such it is included in most graphical Objective-C programs, including iTunes, Safari, Pages, Keynote, and many other widely used applications from Apple and third parties. The Objective-C methods used in the attack are given in Table 1. We extended the framework that Schuster et al. [31] used to create the COOP chains to account for the differences between C++ and Objective-C. The framework uses the SMT solver Z3 [18] to construct a buffer with the constraints defined by the layout of the objects and their required relative offsets to each other. (Recall that typically at least some counterfeit objects overlap.)

For our proof-of-concept exploit, we require a program that contains a memory corruption vulnerability allowing an attacker to place data in the target process as well as overwrite a pointer to an Objective-C instance used during execution. To reliably bypass ASLR, we further require an information leak to disclose the position of the data injected and the location of the instance pointer we override with our own counterfeit object. Since our gadgets are sourced from the AppKit library, this library must also be loaded by the target process. We simulate a suitable vulnerable application by creating an Objective-C program that requires the AppKit library and lets us inject attacker-controlled data in the address space. This data is then interpreted as an Objective-C object, more precisely as our *initial object*, which will start our chain. After this first dispatch the execution is driven entirely by our counterfeit objects.

High Level Overview For our proof of concept we opted to construct a chain that leads to the use of an *invoke gadget* to call an arbitrary function, in this case we chose `system()`. The other gadgets are used to prepare the call by calculating the function address based on import address table (IAT) entries and arranging arguments in memory correctly. After injecting the counterfeit objects into the target process, the chain is started by dispatching a message on the *initial object*, which directs the control flow to the *main loop gadget*. This ML-G dispatches calls to all other gadgets that perform the necessary computations. The chain reads the address of `libsystem!strlen()` from the IAT and adds a precomputed offset to it. The result is then used as the target for the *invocation gadget* (INV-G in COOP parlance [31]). The argument for this call is also located in the attacker-controlled memory and is passed as well. Any precomputed data is passed via fields in the injected counterfeit objects. In Objective-C, an object's fields are also referred to as its *instance variables*.

Initial Object The initial object is the first counterfeit object and is not part of the actual chain. It is designed such that dispatching the corresponding selector on it will enter the ML-G instead of the intended function. Additionally we pass required arguments, in this case the address of the gadgets, as instance variables.

Main Loop At the core of our attack lies the main loop gadget. We use an array-based ML-G (entry 1 in table 1)

[3]In practice, the class struct is oftentimes split into a read/write and a read-only part by the compiler. For brevity, we do not make a distinction between the two here and consider them as one coherent data structure.

#	Method name (AppKit)	Type	Description
1	`[NSTextReplacementNode dealloc]()`	ML-G	main loop
2	`[NSUndoTextOperation affectedRange]()`	LOAD-R64-G	load `rdx` from instance var.
3	`[NSPersistentUIRecord setEncryptionKey:](uint8_t[16])`	R-G	load `rdx` from address `rdx+8`
4	`[NSPanelController stringValue]()`	LOAD-R64-G	load `rcx` from instance var.
5	`[NSMatrix cellAtRow:column:](int64_t, int64_t)`	ARITH-G	`rdx` = `rdx` · `[self+0xf8]` + `rcx`
6	`[NSScrollingScoreKeeper setHoldCount:](int64_t)`	W-G	write `rdx` to instance var.
7	`[NSCustomReleaseData dealloc]()`	INV-G	invoke instance var. as function ptr.

Table 1: Our Subversive-C chain in the standard OS X AppKit library (x86-64) calculates the address of `system()` in `libsystem` and invokes `system("/bin/sh")`. Gadget type names are according to previous work [31].

which iterates over an array of objects and dispatches a constant selector on every entry. Each counterfeit object is an entry in this array. The pseudo code representation of our ML-G is shown in Listing 1; line 5 invokes the selector `release` on every counterfeit object in the injected array. While this particular ML-G is limited to 28 entries in a single array, inserting the ML-G itself again as the 28th entry allows the chaining of more gadgets.

Listing 1: ML-G in NSTextReplacementNode dealloc.

```
1 children = self->children;
2 counter = 0;
3 while (children[counter] != NULL
4     && counter < 28) {
5   [children[counter] release];
6   counter++;}
```

Read Gadget We use two read gadgets (#2 and #4) to load `rcx` and `rdx` from instance variables. As these are *argument registers*, they are guaranteed to remain unaltered by `msgSend`. We load `rdx` with the address of the IAT entry of `strlen()` and `rcx` with the offset between `strlen()` and `system()` in `libsystem`.

Read Gadget with Dereference As we only assume the address of the AppKit module to be given, the address of `system()` in `libsystem` needs to be calculated dynamically. To this end, we read a pointer to `libsystem` from the IAT of the AppKit module and, in the next step, add a constant offset to it. The gadget we use (entry 3) loads `rdx` with the 64-bit value pointed to by `rdx+8`. As we control the value of `rdx` with gadget #2, we can read from a chosen address here. We use this to load `rdx` with the address of `strlen()` from AppKit's IAT.

Arithmetic Gadget At this point `rdx` and `rcx` contain attacker-controlled values and can be used to calculate the actual address of `system()`. Gadget #5 adds both registers and stores the result to `rdx`.

Store Gadget Due to the semantics of our *invocation gadget* (INV-G) (see next step) we need to store the calculated address of `system()` in a specific instance variable of counterfeit object #7. Thus, the two counterfeit objects corresponding to gadgets #6 and #7 need to overlap: gadget #6 stores the function pointer in `rdx` in an instance variable of its counterfeit object; gadget #7 reads this pointer from an instance variable in its counterfeit object (which maps to the same address) and invokes it.

Invocation Gadget The original purpose of our INV-G (#7) is the invocation of a custom deallocator specified via an instance variable. The argument that is passed is also read from an instance variable. This means we both control the function called and its argument. Here, we use this to execute `system("/bin/sh")`.

5 Mitigating Subversive-C

A key insight of our attack is that it targets data structures specific to the Objective-C runtime, much like COOP targets the C++ specific vtable. Therefore, we build our defense around protecting the integrity of these data structures. Unlike C++ vtables, the data structures used by `msgSend` are mutable which means COOP defenses such as vtable randomization [14] are not suitable to protect the Objective-C runtime against Subversive-C. Instead, we choose to base our defense on message authentication to detect malicious tampering.

We add a message authentication code (MAC) to every sensitive field or data structure in the runtime as shown in Figure 3, and use this MAC to verify the integrity of the data structures before sensitive control flow transfers, i.e., those that indirectly use the contents of the data structures. Every time the runtime changes the contents of one of its structures, it also updates the MAC. Thus, an attacker can no longer alter data structures without needing to update the associated MAC. However, each MAC computation has two inputs: the message (data) to authenticate and a secret key. Without both inputs, a correct MAC cannot be computed. Knowing the secret keys would allow attackers to tamper with runtime data structures, so we store them in a key store which attackers cannot read. We describe the key store in detail in Section 5.4.

In the following, we first describe our different approaches to the stages of method lookup, as they have different requirements (most notably the tolerable overhead). Subsequently, we explain the implementation of our secure key store which protects keys from attackers.

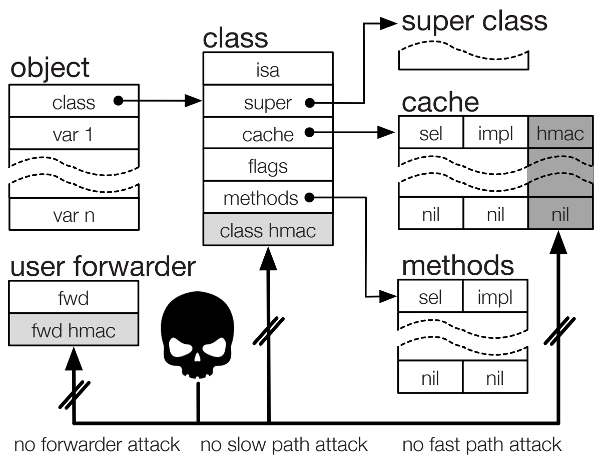

Figure 3: HMACs are used to ensure the integrity of class metadata and caches.

5.1 Securing the Slow Path

To protect the slow path lookup, we repurpose an unused field in the `class` structure to store a MAC (or more precisely a HMAC) as depicted in Figure 3. The hash is populated during class initialization and checked before any class metadata is used for method lookup. If a discrepancy is detected, program execution is aborted with an error message. To compute the hash, we chose the HMAC-MD5 function with the following inputs:

- The **method list** consisting of flags, entry count and an array of `method` structures.
- The **superclass** field to prevent the attacker from modifying the class hierarchy.
- The **flags** field to prevent the attacker from removing the *initialized* bit. An unset *initialized* bit forces the runtime to rebuild the method list (a process which the attacker could tamper with).
- The **isa** field which points to the meta-class of the current class.
- The **address** of the `class` object to uniquely identify the class. A unique identifier is needed to distinguish between similar classes, such as siblings, preventing the attacker from copying the method lists and hash values between them (in such cases, the superclass pointer and flags match).
- A **secret key**—K_{class}—retrieved from a secure key store, which we discuss in more detail in Section 5.4. We use a single global K_{class} for all classes in the application.

Let X be the concatenation of all the elements in the above list except the secret key. We use the following HMAC:

$$H_{\text{class}}(X, K_{\text{class}}) = \text{HMAC}_{\text{MD5}}(X, K_{\text{class}}) \qquad (1)$$

Our choice of the HMAC function is a pragmatic one: HMAC-MD5 is relatively fast, still considered secure [7] (in contrast to MD5), and is available through a library already linked by the Objective-C runtime. Note that the choice of HMAC is a security parameter in our defense; we can replace HMAC-MD5 with any stronger (but likely also slower) MAC in case attacks against HMAC-MD5 appear.

The core assumption of our protection scheme is that the attacker does not know the secret key and hence cannot modify the method list or other metadata used during method lookup without being detected. However, metadata may also change for legitimate reasons. Objective-C is a dynamic language which provides APIs for, e.g., adding classes and methods at run time. We support legitimate changes to metadata via provided APIs by making the change, followed by recomputing the HMAC field.

Note that computing the MAC adds considerable overhead to the slow path lookup (see Section 6.2 for empirical evaluation results). However, lookup results are cached so the slow path is only executed once per (class, selector) pair. Therefore, the steady state program performance remains unaffected. This is also reflected in the implementation of the runtime: the fast path consists of highly-optimized assembly code while the slow path is simply written in C.

5.2 Securing the Fast Path

We protect the fast path in a manner similar to the way we secure the slow path. We implement an authentication mechanism for cache entries that detects any tampering. However, in our practical experiments, we have encountered applications, e.g., iTunes, that modify cache entries directly, i.e., writing to the entries in memory instead of using the runtime API, in much the same way an attacker could tamper with the cache. Therefore, we must allow changes to the cache originating outside the runtime and make sure we detect them and fall back to the slow path.

We implement this by extending the fast path lookup algorithm with an additional authentication step for *cache hits*, as shown in Figure 4. This additional step computes the MAC of the cache entry and checks it against the MAC stored inside the entry. If the hash matches, the algorithm continues normally. Otherwise, the algorithm considers the authentication failure as a *cache miss*. We also modified the runtime to update the stored MAC on changes to a cache entry.

Each cache entry contains two pointers: the selector and the method implementation pointer, as shown in Figure 3. Using these pointers as the MAC input ensures that the attacker cannot modify existing cache entries or add new ones. However, the attacker could still copy entries between caches for different classes, and we wouldn't be able to detect this. Therefore, we add a third pointer to the MAC input: the pointer to the class that owns the

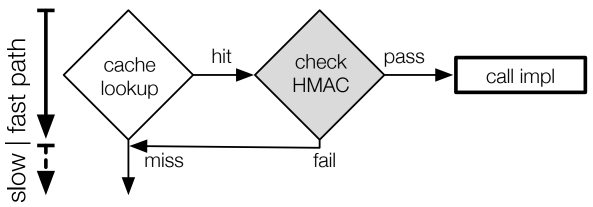

Figure 4: Fast path secured with MAC integrity check.

cache. This prevents the attacker from copying valid cache entries between classes, as each cache entry is now associated with a class.

Unlike the slow path, performance is critical on the fast path and every additional instruction can have a significant impact. Therefore we selected a MAC that we can implement in as few assembly instructions as possible, and easily integrate into the existing cache lookup algorithm. The NH hash function family used in UMAC [8] meets our performance requirements, so we use a modified version of NH as part of our MAC:

$$H_1(X,K) = \sum_{i=0}^{i\leq 2}(X_L[i] + K_L[i]) * (X_H[i] + K_H[i]) \quad (2)$$

where:

- $X = (class, sel, imp)$ is the 192-bit concatenation of the three pointers to hash: the class pointer, the selector and the method implementation.
- $K = (K_0, K_1, K_2)$ is the 192-bit secret key retrieved from the key store.
- $X_L[i]$, $K_L[i]$, $X_H[i]$, and $K_H[i]$ are the low and high 32-bit words of the i^{th} element of X and K, respectively.

The H_1 function has very low collision probability, but is vulnerable to known plaintext attacks (given a large enough sample of $H_1(X,K)$ outputs and their corresponding X inputs, the attacker can compute the K), and therefore insufficient to use as a MAC. UMAC strengthens NH against these attacks by XORing its result with a random number produced by applying a pseudo-random function (PRF) to a nonce.

Using a strong PRF in this case would take too many processor cycles, however, so we use a faster alternative in the form of a fixed-size table T of random 64-bit numbers. We generate this table at program load time, and store it in memory securely as described in Section 5.4. To compute the 64-bit HMAC of a cache entry, we compute H_1, use its output to index into T, and use the resulting value as the output. To simplify indexing, we always allocate T as a table of size $|T| = 2^N$. To compute the index we truncate the output of H_1 to 32 bits and then use the highest N bits. The final form of the HMAC becomes:

$$H_{\text{cache}}(X, K_{\text{cache}}) = T[H_{1[31:(31-N)]}(X, K_{\text{cache}})] \quad (3)$$

5.3 Securing the Forward Handlers

There is another attack vector that the attacker can use during message dispatch: the user forwarder pointers (one for regular message dispatch and one for calls that return structures). The application can legitimately set these pointers using an API call, and many applications use this feature. We prevent attackers from modifying the two pointers by associating a HMAC with each pointer. The runtime updates the HMAC whenever it changes one of the pointers, and checks the HMAC before calling any of the handlers. We once again use a helper function:

$$H_2(fwd, K) = (fwd_L + K_L) * (fwd_H + K_H) \quad (4)$$

that combined with the table T gives us the HMAC:

$$H_{\text{fwd}}(fwd, K_{\text{fwd}}) = T[H_{2[31:(31-N)]}(fwd, K_{\text{fwd}})] \quad (5)$$

5.4 Secure Key Store

Our defense must keep several pieces of information secret to attackers: the HMAC keys—K_{class}, K_{cache}, and K_{fwd}—and the random table T. Discovering these values would allow the attacker to forge the HMAC values and bypass our defenses. It is therefore critical that we prevent attackers from disclosing or guessing these values.

To hide secrets from attackers, we rely on a security primitive known as XoM. This construct allows us to map virtual memory pages in memory so that they will generate a segmentation violation if accessed by anything other than the CPUs instruction fetch unit. Embedding secret values inside such pages allows the runtime to retrieve the values using function calls, while at the same time preventing attackers from reading the pages using direct information leaks. As outlined in Section 3, our threat model assume that one of the available XoM implementations [6, 9, 13, 21] has been deployed on the Objective-C runtime.

We store every secret value inside an execute-only accessor function that returns that secret value when called (the value itself is embedded in the body of the accessor). Additionally, the attacker cannot call the accessor, since that would require hijacking the control flow of the program.

Using one accessor per 64-bit secret value would increase memory usage significantly (we would need an 11-byte accessor for every 8-byte secret, producing a memory overhead of 37.5%), so we take another approach. We store the keys along with T inside a read-only memory region allocated at a random memory address (chosen randomly when calling `mmap`), then store a pointer to this region inside an accessor. To access the table, the runtime calls the accessor to get the pointer, then accesses T using a regular memory read.

To simplify our implementation and reduce the number of accessor calls, we store the HMAC keys as extra cells (one per every 64 bits of key) inside the table T and perform a single accessor call to get the keys and table pointer. This lets the runtime retrieve all secret values using a single accessor call on the fast path, as opposed to one or more per key and then one for the table.

6 Evaluation

In the following, we discuss evaluation results related to the security and performance of our proposed defense.

6.1 Security

We evaluated the effectiveness of our defense using the proof-of-concept Subversive-C exploit described in Section 4. Our hardened runtime is a drop-in replacement which lets us keep all other parameters the same. Thus we can be confident that any differences during program execution are caused by our defense. When running our original attack without any adaptations, the program terminates either due to failing pointer checks (in most cases) or integrity checks. The reason for this is that our original attack does not generate all data structures touched by the integrity checks, but rather the bare minimum necessary to exploit message dispatch. Therefore some dereferenced pointers stay uninitialized. Even if an "accidental pointer" references valid memory, the actual integrity check fails due to the mismatch between computed and stored values.

Next, we extended our attack to generate all data structures that are needed for metadata verification, i.e., all structures that act as input for the HMAC. The easiest way to do so is to copy and then modify existing class structures. However, we were unable to compute the correct HMAC values used to secure the contents of both the cache and the method list. This left us with guessing the right value as the only remaining choice, which is difficult since we need to guess correctly for every counterfeit object in the chain. An incorrect guess for any object leads to detection and program termination.

In both cases Subversive-C is detected before any attacker-controlled code is executed. More specifically, program execution is aborted on the first message that is dispatched to a counterfeit object. As expected, we can create (valid) empty caches, or use the fallback mechanism of the cache protection which triggers a slow path lookup whenever the cache integrity check fails. Creating valid cache entries is difficult due to the keys used in the HMAC being inaccessible to the attacker. With the cache secured, we can try to forge the HMAC for the method list. Here we face even stronger security guarantees since we need to forge HMAC-MD5. Again the attacker lacks the knowledge of the input keys which are protected by the secure key store.

The third way to gain code execution would be to overwrite the forward handlers. However, even with an arbitrary write primitive to allow modifications of these handlers, this will not work. They are protected and the attacker again lacks the secret keys to generate valid handler entries.

We therefore conclude that our hardened runtime probabilistically detects and prevents Subversive-C exploits.

6.2 Performance

Since there is no standard set of Objective-C benchmarks, we compiled the following list of programs to evaluate the performance of our hardened runtime:

- **Dispatch** (micro) invokes a dynamically dispatched (and empty) method in a tight loop. The empty method is invoked 10^8 times.
- **Fibonacci** (micro) computes the 35th Fibonacci number using naive recursion.
- **Sorts**[4] (micro) uses different sorting algorithms (merge, quick, bubble, heap, insertion, selection, and the Objective-C library sort) to sort integer arrays of size 10^4. We combine the running times of all algorithms for our purpose.
- **XML parser**[5] (application) parses and creates song objects from XML data (100 or 1000 entries) using the `NSXMLParser` class [4] from the Objective-C standard library.
- **iTunes play** (application) plays a 5 second audio clip and closes iTunes.
- **iTunes encode** (application) converts a 4 minute song in MP3 format (7 MB) to M4A (7.6 MB) using the AAC encoder provided by iTunes.
- **Pages PDF** (application) exports a 100 page document (270 KB) to PDF (327 KB) in Pages (Apple's word processor).

When reporting results we average over 100 and 10 runs for micro benchmarks and application benchmarks, respectively. We automate the application benchmarks using AppleScript [3] which increases the consistency of our results and allows us to interact with real-world applications. Our hardened runtime is based on the Objective-C runtime version 532.2 (x86-64), which we use as the baseline for performance comparison. Experiments were conducted on an iMac 2.8 GHz Intel Core i7 with 8 GB memory running OS X Yosemite (10.10.5) and the latest versions of iTunes and Pages. In addition, we ran each benchmark with an instrumented version of our runtime to count the number of times the general dispatch function `msgSend` is invoked. Table 2 reports the results of our experiments. Note that the reported numbers do not include the overhead for the defenses assumed in Section 3.

The goal of the Dispatch benchmark is to give us an upper bound for the overhead incurred by our hardened

[4] The Sorts benchmark [24] was developed by Jesse Squires.

[5] XML parser is an adaptation of a benchmark from Apple [5] that compares the performance of XML parsing libraries on iOS.

Benchmark	`msgSend` calls	Calls/ms	Overhead
Dispatch	10,000,000,215	190583	106.46 %
Fibonacci	2,986,070,515	173527	88.66 %
Sorts	13,329,480,611	82597	34.54 %
Average (micro)		148902	**76.55 %**
XML-100	7,940,898	6475	2.81 %
XML-1000	78,119,698	6386	1.97 %
iTunes play	8,592,257	1667	0.37 %
iTunes enc.	114,948	29	1.82 %
Pages PDF	78,691	46	0.75 %
Average (application)		2921	**1.54 %**

Table 2: `msgSend` invocation counts and overheads.

runtime. This is realistic since the benchmark does no real work and just calls an empty method repeatedly. This puts maximum pressure onto the message dispatch mechanism which is the only part of the runtime affected by our protection scheme. Using the data from Table 2 we conclude that the maximum slowdown is bounded by 2x.

The Fibonacci benchmark mainly executes recursive method calls plus an add operation and some control flow to terminate recursion. Note that we mean dynamically dispatched call, i.e., calls dispatched via `msgSend`, whenever we write method call in this section. Standard C function calls are valid in Objective-C, but do not go through `msgSend`. Therefore our defense does not reduce the performance of regular calls to C functions.

The Sorts benchmark is implemented in a way that leads to a high number of `msgSend` calls. Rather than using plain integer arrays, it uses Objective-C collections that require boxing of the integer numbers they store. So what normally is a simple array access becomes two method calls: one to index the collection and one to unbox the integer for comparison. The benchmark results reflect this accordingly. To back our claim we modified the benchmark to use plain integer arrays, replacing `NSMutableArray` with (`int arr[], int len`). As expected, the difference in running times then falls into the range of measurement noise ($< 1\%$).

At this point we want to draw attention to the relation between `msgSend` calls per millisecond and the reported overhead. For compute-intensive programs it is directly proportional. In other words: the more real work a program does, the smaller the overhead.

With the second set of benchmarks we want to demonstrate that although overhead for individual micro benchmarks is considerable, it is insignificant in practice. Especially for interactive applications like iTunes and Pages there is no perceivable slowdown during normal use. For the benchmarks iTunes play and Pages PDF the reported overhead is in the range of measurement noise. Our explanation is that Objective-C is mostly used to implement an application's logic and user interface while core functionality (playing and encoding music files, exporting to PDF) is provided by C libraries. Hence, we incur little to no overhead on those activities. The only time an end user experiences additional delay is during program startup. Table 3 quantifies this delay.

Benchmark		HelloWorld	iTunes
Startup	Base	35	1020
	Hardened	107	1478
Overhead	**Total**	**72**	**458**
	Random table	43	43
	Integrity checks	29	415

Table 3: Startup times and overhead in milliseconds.

We measured the running time of a simple HelloWorld program and the startup time of iTunes both with our baseline and hardened runtime. The total startup overhead for HelloWorld is 72 ms, whereof 43 ms are attributed to seeding the random table which aids the implementation of the secure key store. The remaining 29 ms are spent to populate and check hashes of 280 core classes, e.g., `NSObject`, which are eagerly initialized by the runtime. The time needed to seed the random table depends linearly on the size of the table. In our implementation the table holds 1 million keys resulting in 8 MB total size. The size of the table can be adjusted to adhere to an application's security and memory constraints.

For iTunes the total startup overhead is about half a second. This is due to iTunes being a complex application initializing roughly 2000 Objective-C classes during startup. Considering typical application usage patterns we argue that this is acceptable since there is no further perceived slowdown during continued use.

7 Discussion

In Section 5.4, we presented our approach to securing the key store against leaks: we store its contents at a random address in memory, then store the address as a pointer inside XoM. Since the pointer is stored in a single non-readable location in memory, attackers cannot use an information leak attack to locate the table. However, this approach could expose the table to attackers in some other way, e.g., probing all memory pages one by one to find the table. However, probing attacks would be difficult for two reasons. First, locating all readable virtual memory pages is difficult, assuming attackers cannot install a signal handler or obtain a virtual memory map for the program. Second, to identify the table T in memory, attackers would need to distinguish between randomly-generated bytes and proper program data. Therefore, the barriers to attackers locating T are high. Choosing whether to store the random table in execute-only or readable memory presents a potential security vs. memory usage trade-off. Storing T directly in XoM provides guaranteed secrecy, at the cost of an extra 37.5% memory usage for the table. We therefore leave this decision to system developers.

Side-channel attacks are another potential class of attacks against the key store, or more specifically against the table T. For example, attackers could derive the indices used to access the table, and therefore the values of H_1, by measuring the externally-visible metrics (such as cache misses or CPU cycles) while the runtime performs its integrity checks (similar attacks have been demonstrated on cryptographic functions [41]). If such attacks prove feasible and likely, the same defenses that protect cryptographic algorithms can also be applied to our key store [12].

One other interesting mitigation is object layout randomization. In the runtime, the offsets of instance variables from the start of an object are dynamically defined when its class is loaded. The Objective-C language puts no constraints on the order of variables inside an object, i.e., there is no requirement that they be in the same order that they appear in the source code. Therefore, it would be possible to randomize the object layout. This would not defend against an attacker who can read all of memory, but would make it harder to inject counterfeit objects.

8 Related Work

The work on exploitation and exploit mitigations is extensive. Due to the page limitation, we refer the interested reader to Szekeres et al. [39] and focus on recent, closely related work on attacks and defenses.

Attacks Our work is inspired by the recently published COOP technique [14, 31]. COOP itself is but one of a series of exploitation techniques that are able to bypass coarse-grained CFI policies [10, 16, 23, 32]. By virtue of exploiting the dynamic dispatching mechanisms, both Subversive-C and COOP-style attacks are not stopped by randomization-based defenses that have been widely studied in the last years [26]. RILC is another related exploitation technique [28, 40]. Whereas Subversive-C and COOP reuse dynamically bound methods, RILC reuses dynamically linked functions in the procedure linkage table such as those in the C standard library. Despite the name, RILC applies to other libraries than `libc` [37].

Defenses MoCFI [15] was designed to protect Objective-C code running on iOS for ARM.[6] MoCFI maintains a shadow stack to enforce that a return targets its original caller. Further, forward indirect branches must follow a valid CFG path calculated by means of static analysis. However, Subversive-C circumvents these protections: (1) it never violates call-return matching, and (2) it dispatches all malicious function calls via `msgSend` which resembles a valid CFG path. Further, MoCFI's protection of the `msgSend` selector and class metadata do not prevent Subversive-C since we do not corrupt selectors and avoid changing class structures in ways that MoCFI would detect. Specifically, MoCFI ensures that the class or superclass pointer for each object is known and prevents creation of entirely new classes at run time. However, MoCFI must allow new class structs, where only the superclass pointer is known to MoCFI. As a result, we can construct Subversive-C attacks that use valid superclass pointers or alter the method lists of existing classes. We stress that MoCFI and our novel defense complement each other and can prevent a broader range of attacks when used in concert.

[6]Our research uses but is not specific to x86-64 hardware.

CFR [29] is a compiler-based CFI implementation for Objective-C code on iOS. Unlike MoCFI, which protects returns using a shadow stack, Control-Flow Restrictor (CFR) enforces a purely static policy for all indirect branches. Since CFR does not place any particular restriction on calls dispatched via `msgSend`, CFR does not stop Subversive-C attacks but could complement our defense just like MoCFI. CFR does support programmer-inserted annotations to further constrain the CFG which could potentially prevent our attack; doing so requires manual effort and may lead to errors that prevent programs from running correctly.

Readactor++ [14] is the first randomization-based defense which thwarts COOP attacks by randomizing and booby trapping C++ vtables. Due to the differences in dispatching mechanisms, the concepts behind Readactor++ does not generalize to prevent Subversive-C exploits. For example, vtables are immutable and can be hidden using XoM whereas Objective-C class metadata is mutable which is why we opted to use HMACs instead.

CPI [25] separates regular data from control data which thwarts Subversive-C exploits. CPI relies on static analysis to identify sensitive data which is more challenging for Objective-C code than C and C++ code. It also requires software-fault isolation or hardware segmentation to resist memory probing attacks [19].

Recently, van der Veen et al. [42] presented a purely binary-based defense against COOP, which breaks data flows between vfgadgets through argument registers. Their method enforces a CFI policy derived via static code analysis that limits the vfgadget space available to an attacker, thus making attacks harder. As our example exploit relies on data flows through argument registers, it would be thwarted by this defense. However, we note that Subversive-C—and also COOP in general—does not inherently require register-based data flows, as attackers can potentially fall back to leveraging overlapping counterfeit objects only or passing data via scratch areas.

Similar to our defense, CryptoCFI [27] uses HMACs to protect sensitive pointers. CryptoCFI computes cryptographically secure HMACs using special AES instructions on the latest Intel x86 processors. Although a direct comparison is not possible, the overheads of using this defense is likely far higher than ours and requires that part of the SIMD register file be reserved for CryptoCFI.

9 Conclusion

This paper presented Subversive-C which is the first whole-function reuse attacks abusing the `msgSend` feature of Objective-C. Our attack shows that COOP-style attacks which are far harder to prevent than ROP-style code-reuse, are not limited to C++ code. We discuss the intricacies of Objective-C message dispatch and how to utilize them for our attack. Specifically, we describe an attack targeting the AppKit (x86-64) library for OS X, which is a core building block for many popular applications. Finally, we present a practical defense against Subversive-C and show that it imposes a negligible performance overhead when protecting real-world applications.

Acknowledgments

We would like to thank David Chisnall, our shepherd Randal Burns, and the anonymous reviewers for their valuable input. Fabian Kammel helped with the implementation of the attack outlined in Section 4.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-15-C-0085, and FA8750-10-C-0237, by the National Science Foundation under award number CNS-1513837, by the ERC Starting Grant No. 640110 (BASTION) as well as gifts from Mozilla, Oracle and Qualcomm. In addition, this work was supported in part by the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union's Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security 13* (2009).

[2] ALEPH ONE. Smashing the stack for fun and profit. *Phrack Magazine 7* (1996).

[3] APPLE INC. AppleScript language guide. `https://developer.apple.com/library/mac/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html`, 2015.

[4] APPLE INC. NSXMLParser class reference. `https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSXMLParser_Class`, 2015.

[5] APPLE INC. XMLPerformance on iOS. `https://developer.apple.com/library/ios/samplecode/XMLPerformance/Introduction/Intro.html`, 2015.

[6] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[7] BELLARE, M. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology - CRYPTO 2006*, C. Dwork, Ed., vol. 4117 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 602–619.

[8] BLACK, J., HALEVI, S., KRAWCZYK, H., KROVETZ, T., AND ROGAWAY, P. UMAC: Fast and secure message authentication. In *Advances in Cryptology—CRYPTO* (1999).

[9] BRADEN, K., CRANE, S., DAVI, L., FRANZ, M., LARSEN, P., LIEBCHEN, C., AND SADEGHI, A.-R. Leakage-resilient layout randomization for mobile devices. In *Symposium on Network and Distributed System Security (NDSS)* (2016), NDSS.

[10] CARLINI, N., AND WAGNER, D. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium* (2014).

[11] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[12] CRANE, S., HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Thwarting cache side-channel attacks through dynamic software diversity. In *Symposium on Network and Distributed System Security (NDSS)* (2015).

[13] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[14] CRANE, S., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., SUTTER, B. D., AND FRANZ, M. It's a TRAP: Table randomization and protection against function reuse attacks. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[15] DAVI, L., DMITRIENKO, A., EGELE, M., FISCHER, T., HOLZ, T., HUND, R., NÜRNBERGER, S., AND SADEGHI, A.-R. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *NDSS* (2012).

[16] DAVI, L., LEHMANN, D., SADEGHI, A.-R., AND MONROSE, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium* (2014).

[17] DAVI, L., SADEGHI, A.-R., AND WINANDY, M. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2011).

[18] De Moura, L., and Bjørner, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008).

[19] Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., and Okhravi, H. Missing the point: On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[20] Frantzen, M., and Shuey, M. StackGhost: Hardware facilitated stack protection. In *USENIX Security Symposium* (2001).

[21] Gionta, J., Enck, W., and Ning, P. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *ACM Conference on Data and Application Security and Privacy (CODASPY)* (2015).

[22] Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)* (2014).

[23] Göktas, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium* (2014).

[24] Jesse Squires. Objective-c sorts. `https://github.com/jessesquires/objc-sorts`, 2014.

[25] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer integrity. In *USENIX Security Symposium* (2014).

[26] Larsen, P., Homescu, A., Brunthaler, S., and Franz, M. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy (S&P)* (2014).

[27] Mashtizadeh, A. J., Bittau, A., Boneh, D., and Mazières, D. CCFI: Cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[28] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine 11* (2001).

[29] Pewny, J., and Holz, T. Control-flow Restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)* (2013).

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

[31] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., and Holz, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[32] Schuster, F., Tendyck, T., Pewny, J., Maass, A., Steegmanns, M., Contag, M., and Holz, T. Evaluating the effectiveness of current anti-ROP defenses. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2014).

[33] Serna, F. J. The info leak era on software exploitation. In *BlackHat USA* (2012).

[34] 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).

[35] Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)* (2004).

[36] Siebert, J., Okhravi, H., and Söderström, E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[37] Skowyra, R., Casteel, K., Okhravi, H., and Zeldovich, N. Systematic analysis of defenses against return-oriented programming. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2013).

[38] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[39] Szekeres, L., Payer, M., Wei, T., and Song, D. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[40] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V. W., and Ning, P. On the expressiveness of return-into-libc attacks. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2011).

[41] Tromer, E., Osvik, D. A., and Shamir, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* (2010).

[42] van der Veen, V., Göktas, E., Contag, M., Pawlowski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., and Giuffrida, C. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE Symposium on Security and Privacy (S&P)* (2016).

[43] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical control flow integrity & randomization for binary executables. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[44] Zhang, M., and Sekar, R. Control flow integrity for COTS binaries. In *USENIX Security Symposium* (2013).