PSiOS: Bring Your Own Privacy & Security to iOS Devices

Tim Werthmann[1], Ralf Hund[1], Lucas Davi[2,3], Ahmad-Reza Sadeghi[2,3] and Thorsten Holz[1]

[1]Ruhr-Universität Bochum, Germany [2]Intel CRI-SC at TU Darmstadt, Germany
[3]Technische Universität Darmstadt, Germany

ABSTRACT

Apple iOS is one of the most popular mobile operating systems. As its core security technology, iOS provides *application sandboxing* but assigns a generic sandboxing profile to *every* third-party application. However, recent attacks and incidents with benign applications demonstrate that this design decision is vulnerable to crucial privacy and security breaches, allowing applications (either benign or malicious) to access contacts, photos, and device IDs. Moreover, the dynamic character of iOS apps written in Objective-C renders the currently proposed static analysis tools less useful.

In this paper, we aim to address the open problem of *preventing* (not only detecting) privacy leaks and simultaneously strengthening security against runtime attacks on iOS. Compared to similar research work on the open Android, realizing such a system for the closed-source iOS is highly involved.

We present the design and implementation of *PSiOS*, a tool that features a novel policy enforcement framework for iOS. It provides fine-grained, application-specific, and user/administrator defined sandboxing for each third-party application *without* requiring access to the application source code. Our reference implementation deploys control-flow integrity based on the recently proposed MoCFI (Mobile CFI) framework that only protects applications against runtime attacks. We evaluated several popular iOS applications (e.g., Facebook, WhatsApp) to demonstrate the efficiency and effectiveness of *PSiOS*.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

iOS; Application Sandboxing; Software Security

1. INTRODUCTION

Smartphones and tablet computers are becoming ubiquitous, and the sales figures for both types of devices are growing rapidly. Probably the most important factor behind this popularity is the availability of a large number of mobile applications, ranging from simple games over messaging apps to office applications. Consumers can easily install these apps via so-called *app stores* and then use them. Privacy and security concerns arise because an application can also access personal/sensitive information on the device, such as for example contact details, messages, location information, or private photos. Current mobile operating systems are following slightly different approaches to protect a user's personal information and we focus in this paper on the closed-source Apple iOS since it is after Android the most popular mobile OS on the market. Further, most security extensions for smartphones target the *open* Android, while security extensions for the *closed* iOS are rarely available.

iOS assigns a generic sandboxing profile to every third-party application allowing access to contacts, location, photos, recent searches, and device information. This design decision contradicts the least-privilege principle, and easily allows an application (either benign or malicious) to access sensitive information. Developers must adhere to certain guidelines for their applications to be accepted in the Apple App Store [7]. According to these guidelines, apps must ask for the user's permission to read out sensitive data. In the recent past, however, several (legitimate) apps were detected that abused these privileges and for instance uploaded the complete contact list to the app developers without the user's consent [26]. These incidents even lead to an investigation of the US Congress regarding data collection practices on mobile devices [21].

To mitigate this problem, one of the first approaches is to *detect* privacy leaks in iOS apps by using *static* analysis, as recently proposed by Egele et al. [14]. Their static analysis tool, called PiOS, generates the control-flow graph of an application to identify code regions that potentially steal sensitive information. PiOS focuses on a *static* analysis approach and we identified shortcomings of the actual analysis phase which we detail in Section 8. Up to now, there is no solid privacy framework for iOS that enables a user to *protect* his personal information. Compared to Android, for which such tools have been proposed [19, 24], our main challenges concern the proprietary, closed-source nature of iOS as well the usage of the programming language Objective-C. The latter significantly complicates the implementation due to the highly dynamic nature (e.g., dynamic binding and

Objective-C class clusters). Moreover, while on Android all applications access the main phone functionalities through the Java Android API, iOS apps may use either Objective-C, standard C/C++, or direct system calls to use the main phone services. This raises a significant technical challenge, since all possible access techniques have to be considered to ensure accurate access control on privacy-related information and operations.

Contribution.

In this paper, we aim to address the open problem of not only detecting privacy leaks for iOS apps, but actually *preventing* them. The key idea of our approach is to assign specific sandboxing profiles to each application to enforce a given fine-grained privacy policy. Such a profile may either be defined by a user at installation time, or centrally provided by a system administrator or an enterprise. Logically, we generate a protection layer between applications and the iOS Objective-C Runtime environment. Further, we monitor applications at execution time and ensure that they only perform actions that adhere to the given sandboxing profiles. Our solution operates *directly* on the application binary, and hence, neither requires recompilation nor access to the source code, which enables enforcement of policies for arbitrary applications.

We have implemented a fully-working and efficient prototype of our framework in a tool called *PSiOS* that is based on MoCFI [11], a tool that enforces control-flow integrity on iOS devices. *PSiOS* enables user-driven and fine-grained application sandboxing: on the one hand, a user can dynamically update sandboxing profiles, without the need to recompile or reinstall the application. Hence, end-users can easily revoke or assign privileges. On the other hand, our framework enables very fine-grained policies in which the user or system administrator can precisely specify which privileges are assigned to an application. This is possible, since our sandboxing profiles cover the *entire* Objective-C runtime and allows argument validations for each API call. Since our framework is based on a complete control-flow integrity tool, we also prevent attackers from exploiting vulnerabilities in the application code to hijack its assigned rights [20, 22]. We demonstrate that *PSiOS* effectively prevents privacy breaches by testing our tool with SpyPhone [23], an iOS app specifically designed to steal sensitive information from an iOS device. Furthermore, we successfully instrumented many real-world, complex applications like Facebook or WhatsApp and performance measurements indicate that the overhead introduced by our tool is reasonable.

Note that our approach differs from Apple's recently introduced entitlement keys (i.e., permissions). Specifically, Apple provides 25 keys to confine the privileges of apps [6]. However, these keys are specified by the app developer or directly by Apple. Hence, neither the end-user nor an enterprise can apply custom sandboxing policies to their apps. Instead, one needs to rely on Apple to apply the appropriate entitlements to each app, while malware writers will avoid to confine their apps for obvious reasons. Moreover, as we will elaborate in Section 2.2, these entitlements are enforced on the basis of the built-in iOS sandboxing framework, which (in contrast to *PSiOS*) cannot enforce fine-grained sandboxing rules.

Independent from our research work on *PSiOS*, Apple has very recently introduced privacy settings options (starting from iOS 6) where end-users can disable or enable access to private information on an app-by-app basis [8]. We believe that this new feature is a step in the right direction since iOS devices suffer from lack of privacy protection. However, we stress that our tool *PSiOS* does not only cover the same access rules, but also features argument validation (e.g., to allow access to a subset of private information), enables fine-grained access control beyond access to private information (e.g., any system call an application may invoke), and at the same time prevents runtime attacks.

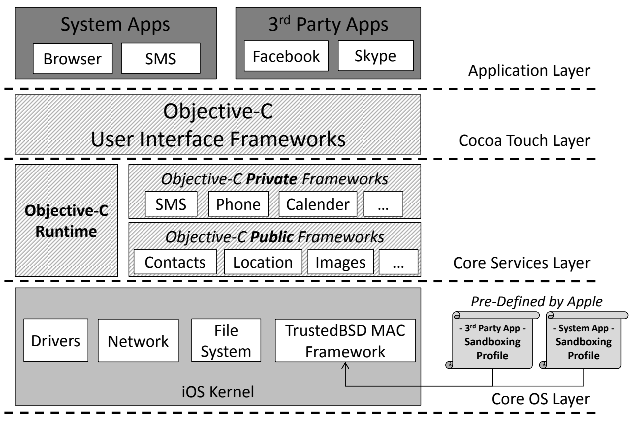

Figure 1: iOS Software Architecture

2. BACKGROUND AND PROBLEM DESCRIPTION

In this section, we recall the basics of the iOS architecture and describe why iOS sandboxing suffers from severe security and privacy problems.

2.1 iOS Background

Apple devices such as iPhone, iPod touch, and iPad are based on the iOS operating system that provides several security features, e.g., mandatory code signing, data encryption, or memory randomization. In particular, iOS enforces *application sandboxing* to isolate applications from each other, i.e., an application cannot access files in another application's directory. Moreover, it constrains third-party applications from accessing the underlying operating system kernel.

Figure 1 shows an abstract representation of the iOS software architecture. Note that the original iOS architecture also includes a Media layer, which we did not include for the sake of brevity. Basically, iOS features four software layers that are relevant for our analysis: (1) an application layer, (2) the so-called Cocoa Touch layer which mainly provides objects for the application display, (3) the Core Services layer which provides frameworks to access main phone facilities, and (4) the Core OS layer (the iOS kernel) that provides basic OS facilities such as a system call wrapper, device drivers, and the file-system.

Objective-C Runtime.

iOS applications and the main iOS system libraries are implemented in the object-oriented language Objective-C. Since iOS defers many decisions from compile-time to runtime, a *runtime system* is required in order to use Objective-C [4]. The Objective-C runtime is linked to every process and interacts with the so-called Objective-C *frameworks*. A framework is Apple's notion for a directory containing a shared library along with resources to support the framework (e.g., images, header files). The Objective-C core frameworks are provided in the Cocoa Touch and Core Services layer, while iOS distinguishes between *public* and *private* frameworks. The private frameworks are only accessible by system applications, while the public frameworks can be accessed by every third-party application. Of particular interest are the frameworks located in the Core Services layer as they provide access to main phone facilities (such as Location, SMS, Calendar, Contacts, etc.).

2.2 Problem Description

iOS sandboxing is realized by a kernel module which has been adopted from the TrustedBSD kernel. This module mediates and validates every system call and its arguments according to sandboxing profiles already *pre-defined* by Apple. As already mentioned, iOS assigns a generic sandboxing profile to *every* third-party application which enables every application to access the public frameworks and specifically grants access to contacts, location, device information, call history, keyboard cache, recent searches, e-mail account configurations, and photos. Recently, several attacks were reported where applications abused their privilege set to steal, for instance, the user's address book [26]. Moreover, whenever an application is exploited by a runtime attack, the adversary can misuse the application's privileges to steal sensitive information as well [20, 22].

Note that iOS already supports sandboxing at the kernel-level, but not within the Objective-C runtime. The design decision taken by Apple leads to coarse-grained sandboxing, because the Core OS layer misses the semantics of the Objective-C runtime. Instead of enforcing access-control on a specific API call, iOS has to enforce access-control based on invoked system calls. Hence, it cannot provide a fine-grained access control. In particular, the Core OS layer cannot enforce access control on the main Objective-C constructs such as used classes, objects, variables, and methods, which are extensively used by iOS applications and involve a chain of diverse system calls, files, and memory structures.

Individual iOS sandboxing rules can be bound to mobile apps using *entitlements* [6]. These entitlements assign certain rights to applications, which are in turn enforced by the iOS sandbox described previously. Apple currently supports 25 different OS X entitlements for that purpose, of which only a subset is available on iOS. However, the system has a major drawback: entitlements are requested by the developer and included in the digital signature of the application by Apple's app store. This means that they cannot be changed afterwards by the end user since this would break the entire signature.

3. HIGH-LEVEL IDEA

In this paper, we address the mentioned security and design weaknesses of the current iOS sandboxing realization (see Section 2). In particular, we aim towards a framework that allows access-control for the Objective-C runtime and the enforcement of the least-privilege principle. Note that realizing such a system for iOS is highly involved, since iOS is closed-source. Hence, we cannot simply replace or extend existing modules as typically performed in recent Android security research proposals, e.g., Kirin [16] or TaintDroid [15] to name a few.

The high-level idea of *PSiOS* (**P**rivacy and **S**ecurity for **iOS** devices) is shown in Figure 2. In contrast to Apple's approach (where sandboxing profiles are generic and pre-defined), *PSiOS* allows a different and user-defined sandboxing profile for each application. Basically, we add a new module that operates between the Application and Cocoa Touch Layer (see Section 2), which we call *policy enforcement* component. As shown in Figure 2, this component mediates every access request to the Objective-C runtime, the frameworks, and the system call wrapper. It enforces access control rules on each access request based on the user-defined sandboxing profiles. Only when the policy has not been violated, we forward the request to its original destination. Note that the current iOS system does not enforce *any* access control mechanism to the Objective-C runtime and frameworks. On the other hand, iOS already enforces access control on system calls, but our approach allows an individual enforcement policy for each iOS application.

To monitor if an application adheres to the given sandboxing profile, we instrument the application so that all access requests are redirected to the policy enforcement. Alternatively, one could directly extend the Objective-C runtime and frameworks with dedicated interfaces that validate whether the caller has the appropriate privileges (similar to the Android permission system [18]). However, the iOS Objective-C runtime and frameworks are closed-source. Hence, extending them directly is infeasible.

Furthermore, the Objective-C runtime operates on the same level as the application code and checks within the runtime could thus easily be bypassed, either intentionally by the application author or by a control-flow attack against a software vulnerability. Our framework does not suffer from this shortcoming since we have full control over the application code. We also protect the application from being compromised by control-flow attacks (such as code injection [2] and return-oriented programming [28]). Otherwise an adversary could leverage such attack techniques to circumvent our instrumentation points and the policy enforcement component. To tackle this problem, we ensure the integrity of the application's execution flow by enforcing control-flow integrity (CFI) which is a general countermeasure to defeat such attacks by validating if an application always follows the application's control-flow graph (CFG). Recently, we introduced MoCFI (Mobile CFI) [11] which enforces CFI on mobile devices, where iOS has been used for the proof-of-concept implementation. We leverage MoCFI for our prototype, but it is noteworthy to mention that in contrast to MoCFI, we replace PiOS [14] with a novel Objective-C analyzer, to tackle shortcomings of PiOS (see Section 8).

4. DESIGN OF PSIOS

In this section we introduce the design of *PSiOS* (**P**rivacy and **S**ecurity for **iOS** devices) which enforces our high-level idea presented in Section 3.

4.1 Architecture

The general architecture of *PSiOS* is depicted in Figure 3. Basically, our design can be divided into three distinct phases: (1) static analysis (*offline*), (2) binary rewriting at *load-time*, and (3) runtime CFI and policy enforcement at

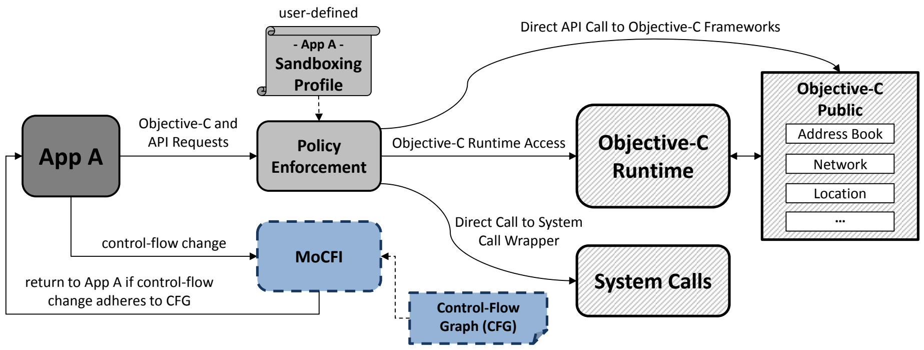

Figure 2: High-Level Idea of *PSiOS*

execution-time. While the static analysis phase needs to be performed only once, the binary rewriting and runtime enforcement phase are performed whenever the application is launched by the user.

The general workflow is as follows: First, we reverse-engineer the application binary by using automated tools to derive the application's structure. In particular, we leverage MoCFI [11] to derive the application's control-flow graph (CFG), which is required to extract all valid execution paths (step 1). Further, we implemented a static Objective-C analyzer that reuses techniques of existing tools such as PiOS [14] and Objective-C helper scripts [13] to identify used Objective-C classes and methods (step 2). Note that we extended these tools to cover all calls to the system call wrapper as well. When the application is launched by the user, we first perform binary rewriting to integrate control-flow checks into the binary (step 3).

Second, we leverage binary rewriting to insert checkpoints into the application that will be reached whenever an application aims to access the Objective-C runtime, the public frameworks, or the system call wrapper (step 4).

At execution-time, we first use a novel runtime Objective-C analyzer that tackles the incompleteness of the static analysis and retrieves important runtime information on Objective-C constructs (step 5). Afterwards, CFI ensures that the control-flow of an application always follows the legitimate paths of the CFG (step 6). Further, for all access requests to the Objective-C environment and the system call wrapper, our policy enforcement component validates if the request adheres to the given policy rules (step 7). In the following, we elaborate in more detail on the individual components and phases.

Static Analysis.

Since iOS applications are encrypted by default, we first obtain the decrypted version of an iOS binary by using a technique called *process dumping* [14], which automatically creates memory snapshots of the unencrypted code at runtime. Afterwards, we use MoCFI to derive the CFG, i.e., to resolve the targets of indirect branches (e.g., indirect calls/jumps, and function returns). In the original design of MoCFI, PiOS [14] is used to resolve Objective-C calls. However, as our experiments revealed, the static analysis performed by PiOS is not always able to retrieve the used Objective-C structures. Other existing approaches to parse Mach-O Objective-C files [13] are outdated and only process a subset of the required constructs. Hence, we developed a novel static Objective-C analyzer which is capable of identifying all relevant Objective-C structures describing classes, methods, and inheritance relationships. Further, our tool provides a more fine-grained analysis than the existing ones, since it completely parses the iOS Mach-O File Header and accurately resolves all calls to the system call wrapper. Moreover, note that the existing tools [14, 13] do not take the dynamic nature of the Objective-C runtime environment into account, which is necessary to enforce fine-grained policy enforcement and handle cases that cannot be resolved statically. The CFG and the Objective-C information are stored in separate configuration files.

Binary Rewriting and Runtime Enforcement.

To preserve the application signature, we perform binary rewriting after the iOS application loader has verified the application signature. For this *PSiOS* employs a rewriting engine to patch all indirect branch instructions with a control-flow check. Furthermore, *PSiOS* rewrites all access requests to the Objective-C runtime to insert checkpoints. After the binary rewriting, our runtime Objective-C analyzer requests crucial runtime information on Objective-C constructs (such as registered parent and child classes, and runtime addresses of invoked methods) which cannot be derived in the static analysis phase. Afterwards, the application starts executing and whenever a checkpoint has been reached, *PSiOS* ensures that the call follows a valid CFG path and adheres to the given sandboxing profile. Our approach ensures that the application cannot escape from the newly established sandbox, which may occur by either directly calling the Objective-C runtime or frameworks, or indirectly, by launching a runtime attack.

Enforcement Policies.

PSiOS supports three different policy enforcement types: *Log*, *Exit*, and *Replace*.

Each enforcement type can be used to achieve a different objective. The *Log* option ensures that all policy violations are recorded by the system. However, the application is allowed to continue executing, but the system's log file can be inspected afterwards. This option is in particular useful when applied in a learning phase (for instance in case of enterprises) in order to help system administrators to identify which Objective-C calls are performed and required by the application. Hence, it facilitates the specification of

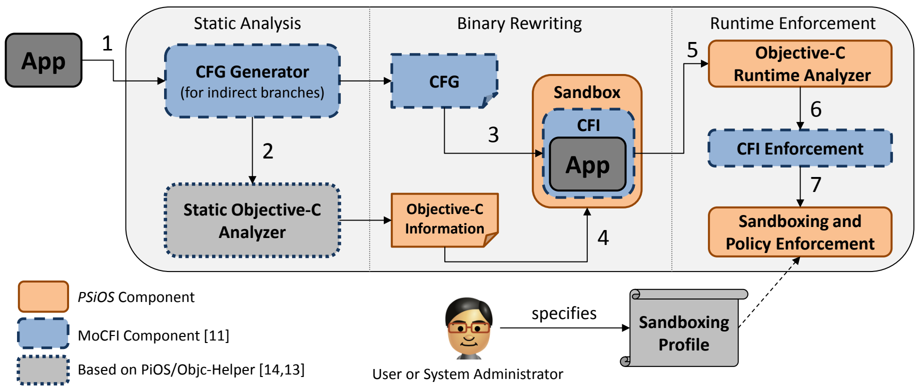
Static Analysis
Binary Rewriting
Runtime Enforcement
App
1
CFG Generator (for indirect branches)
2
Static Objective-C Analyzer
CFG
3
Objective-C Information
4
Sandbox
CFI
App
5
Objective-C Runtime Analyzer
6
CFI Enforcement
7
Sandboxing and Policy Enforcement
PSiOS Component
MoCFI Component [11]
Based on PiOS/Objc-Helper [14,13]
specifies
Sandboxing Profile
User or System Administrator

Figure 3: Architecture of *PSiOS*

sandboxing profiles. The *Exit* option is more restrictive, because it immediately terminates the process whenever a policy violation occurs. Due to usability reasons, we also introduced the *Replace* option which allows the application to continue executing, but *PSiOS* replaces the return values of the Objective-C runtime with shadow data. For instance, if the sandboxing profile prohibits access to the address book, the return value will be either fake contacts or an empty data set indicating to a user that a policy violation occurred.

Our policy enforcement can be used to achieve different goals. For example, an end user can be enabled to specify which mobile app may access which privacy critical resources. A graphical interface can then manage the assignment of policies to applications in a user-friendly way, similar to Android's permission system [18]. Furthermore, policies could also be deployed by a central instance (e.g. an enterprise) in a *bring your own device* environment. This allows the company to mitigate the risk of sensitive corporate data leakage (e-mails, text message, address book entries, etc.) through insecure private applications. Our current prototype only provides the underlying framework for policy enforcement and does not come with a frontend. However, it is straightforward to implement such a component in the future, we focus on showing the feasibility of the approach.

Note that providing complete policies is tedious work. Many tasks can be achieved by using different APIs. For instance, files can be accessed by using a variety of function calls (e.g., through Objective-C, the C standard lib, etc.). Identifying all functions that can be used to access a privacy critical ressource takes considerable effort, but is possible nevertheless.

4.2 Format of Sandboxing Profiles

The sandboxing profile consists of blacklisted Objective-C and API Calls, while each policy rule follows the following XML-based format:

```
<rule type="objc" class="<classname>"
   selector="<selector>" mode="log|exit|replace">
     <arg number="<arg number>" type=
        "int|string|..." operator="=|!|<|>|<=|>="
          value="<argument value>"/>
     <arg ... />
</rule>
<rule type="api" function="<functionname>"
    mode="log|exit|replace">
     <arg ... />
      ...
</rule>
```

The first attributes of an Objective-C rule refer to the class and selector (i.e., the Objective-C method name). If the policy targets an API call, the API call name is stored in the `function` field and the `type` is set to `api`. The `mode` field indicates the enforcement type. The `number` field of an `arg` mode indicates the argument number and the `type` field identifies the argument's type, e.g., `int` refers to an integer. The `operator` indicates the compare method $(=, !, <, >, >=, <=)$ and the `value` field holds the compare value. For better understanding, consider the following Objective-C message:

```
[NSUserDefaults valueForKey:@"
      SBFormattedPhoneNumber"]
```

This Objective-C message initializes a new instance of the `NSUserDefaults` class and uses the `valueForKey:` selector (i.e., method) with the argument `SBFormattedPhoneNumber` to retrieve the device's phone number. Note that this call can be invoked by every application. Hence, a policy rule to strictly deny this Objective-C message would look as follows:

```
<rule type="objc" class="NSUserDefaults"
    selector="valueForKey:" mode="exit" />
```

This rule is always triggered if the class and selector are set to `NSUserDefaults` and `valueForKey:`. If only those messages that use the `SBFormattedPhoneNumber` as method argument should be denied, the rule has to be changed as follows:

```
<rule type="objc" class="NSUserDefaults"
    selector="valueForKey:" mode="exit">
     <arg number="1" type="string" operator="="
          value="SBFormattedPhoneNumber"/>
</rule>
```

This allows us to define fine-grained policy rules on each Objective-C method and API call. Moreover, the argument validation of our policy language supports logical operations such as *AND* and *OR*.

5. BACKGROUND ON OBJECTIVE-C AND CHALLENGES

In this section, we briefly recall the basics of the Objective-C runtime and elaborate on the challenges to be tackled when implementing *PSiOS*. Further, we describe why existing static tools [14, 13] miss important information that is only available at runtime.

In general, Objective-C is an object-oriented, dynamically typed language, and mainly operates on objects. In particular, Objective-C provides three basic constructs for encapsulating data with methods: metaclasses, classes, and objects. Objects are instances of classes, whereby classes are instantiated from their metaclasses, and metaclasses are instantiated from their root's metaclass. Further, every class can have an arbitrary amount of child classes.

Messaging.

A distinct feature of Objective-C is its messaging engine. Instead of calling a method directly, applications send a message to the Objective-C runtime, which deploys a dynamic message dispatcher to interpret the message. The generic syntax of an Objective-C message is as follows:

```
[object method:arg1 param_name1:arg2 ... param_
    nameN-1:argN]
```

It consists of the target object (usually called *receiver*) and the method name along with its parameter names and argument values. In particular, the method name and the parameter names form the so-called Objective-C *selector*. The Objective-C runtime interprets these messages and forwards them to the receiving objects which execute the desired methods.

For better understanding, consider the example Objective-C message used in Section 4.2, where `valueForKey:` forms the selector of the message:

```
[NSUserDefaults valueForKey:@"
    SBFormattedPhoneNumber"]
```

The selector is unique to the runtime system and is used to derive a *unique identifier* of type `SEL` (a compiled selector) that is registered by the runtime system. However, since each object can implement its own version of a method, the same selector might refer to different implementations. Thus, the Objective-C runtime determines the appropriate method implementation based on the class of the receiver at runtime. This technique is referred to as *dynamic binding* and is realized by compiling every Objective-C message into a default call to the Objective-C runtime messaging function `objc_msgSend`, which has the following format: *objc_msgSend(receiver, selector, arg1, arg2, ...)*

Challenge: Due to dynamic binding, a compiled iOS binary will consists of a high number of calls to the generic dispatcher function `objc_msgSend`. Hence, straight-forward disassembling (as performed by IDA Pro) will only output a number of calls to the dispatcher function rather than providing the real target object and method. Although existing tools [14, 13] partly address this problem, they cannot determine the actual implementation of the method since the same selector might reference different implementations.

Class Clusters.

Objective-C class clusters allow the grouping of related or dependent objects under a single public abstract superclass. A typical example is the Objective-C class cluster for numbers: while the *Number* class is the public abstract superclass of the cluster, the *Integer* or *Float* classes are private subclasses of the *Number* class. Hence, the private subclasses are not visible to the developer, who only uses the public *Number* class to instantiate a new number. In turn, the *Number* class will itself perform the correct instantiation of the respective integer or float.

Challenge: The challenge in this context is that private subclasses are not visible at static analysis time, and hence, existing static tools [14, 13] can neither identify malicious behavior through these classes nor provide accurate control-flow information with regard to the control-flow graph.

6. IMPLEMENTATION

We implemented the design of *PSiOS* (see Figure 3) in a prototype that supports iOS version 4.3.2, 4.3.3, 5.0.1, and 5.1.1. The static Objective-C analyzer is realized as a new Python module for the reverse-engineering tool IDA Pro 6.x. For rewriting the application binary and enforcing CFI, we base our implementation on our MoCFI framework [11]. However, we need to extend MoCFI to introduce the policy enforcement and the runtime Objective-C analyzer. In particular, the latter component enables runtime analysis of Objective-C constructs to tackle the challenges mentioned in Section 5. The entire runtime tools are realized as one shared iOS library that is developed in the Objective-C++ language. Since Apple prohibits any user from installing a new shared library, we had to jailbreak our test devices to inject our library to every iOS application and to enable binary rewriting at runtime, as discussed in Section 7.4. In the following, we present selected implementation details of the core components of the *PSiOS* framework, while we refer to [11] for the implementation details of MoCFI.

6.1 Static Objective-C Analyzer

Basically, the static Objective-C analyzer performs binary analysis of an iOS application to identify all implemented and referenced Objective-C classes and selectors. For this we reuse techniques from existing tools such as PiOS [14] and Objective-C scripts [13]. However, we extend the existing tools to resolve all direct calls to the system call wrapper. Note that static analysis of iOS binaries is a challenging task because most internals have not been documented, and required high reverse-engineering efforts on our side.

Analysis of iOS File Header.

In the first step, our analyzer parses the entire iOS file header (Mach-O) header, which is necessary to locate all code and data sections and to identify important side information on used Objective-C constructs and the binding information referenced at runtime by the iOS linker. For this, we examine the so-called load commands included in the iOS Mach-O header [3]. For the sake of completeness, the load commands inspected by *PSiOS* are listed in Table 2 in Appendix A. Of particular interest are the `LC_DYLD_INFO` and the `LC_DYLD_INFO_ONLY` load commands as they provide binding information that is used at load-time by the iOS loader to resolve API calls and Objective-C classes.

Objective-C Classes and Selectors.

After analyzing the iOS header, we identify all Objective-C classes and selectors used by the application. This includes new Objective-C classes the developer implemented himself and classes that have been already pre-defined by the Objective-C runtime. In particular, we reuse the existing tools [14, 13] to record for each call to the `objc_msgSend` dispatcher function the referenced Objective-C class and selector.

API Calls.

Direct API calls to the public frameworks and the system call wrapper can be resolved similarly to imported Objective-C constructs since the symbol section (`__lazy_symbol`) holds 4-byte addresses which identify API calls used in the code. Backtracking the references to the symbol section reveals that each address is referenced by an address in the section `__symbolstub1`[1]. The addresses of the `__symbolstub1` section are then used throughout the code to invoke API calls, which allows us to identify the location and type of all API calls.

6.2 Objective-C Runtime Analyzer

Our novel Objective-C runtime analyzer tackles the incompleteness of the static analysis phase. In particular, it starts its operation after the iOS loader has loaded the application into memory. The basic information our analyzer derives are the *runtime addresses* of all selectors and the missing parent and child classes (which are necessary to support class clustering).

First, we retrieve the runtime addresses of the used selectors by referencing the `__objc_selrefs` section. However, the selectors can be only available as textual strings (rather than runtime addresses). To tackle this problem, we use the string representation of the selector and issue the Objective-C Call `NSSelectorFromString` to retrieve the selector's runtime address. In particular, we create a hashmap *objcHashmap* that stores all selector runtime addresses.

Afterwards, we add all Objective-C classes to the *objcHashmap* by using their runtime addresses. We retrieve the runtime addresses from the `__objc_classrefs` section and in case only the class name has been derived at static analysis time, we use the `objc_getClass` function to retrieve the corresponding runtime address. Note that we complete this step by requesting all registered classes from the Objective-C runtime. This comprises all parent and child classes which allows us to support class clustering and tackles the challenge mentioned in Section 5.

6.3 Policy Enforcement

The policy enforcement is the core component of *PSiOS* as it enforces access control on each Objective-C message and direct call to the system call wrapper.

To initialize the policy enforcement we first use the API call name (derived from the static tools) and leverage the dynamic iOS loader to retrieve the runtime address for each API call. Afterwards, we open and read the sandboxing profile file of the application. Since each rule in the sandboxing profile targets a particular Objective-C selector, class, and API call, we validate which of these constructs are actually used by the application. This can be determined by comparing the constructs of the sandboxing profile to the *objcHashmap* and the derived API calls. Only for those rules that are relevant for the application, we add the class, selector, API call name, and the policy rule to a second hashmap, we refer to as *polHashmap*.

At runtime we enforce a policy check whenever a function call to the Objective-C runtime, the public frameworks, or the system call wrapper occurs. First, MoCFI validates that control-flow integrity is preserved for the call. Afterwards, our policy enforcement checks if the call adheres to the given sandboxing profile.

For all Objective-C messages invoked at runtime, our policy enforcement looks up the class and the selector of the involved message in the *polHashmap*. When the *polHashmap* contains a rule for this message we validate whether the rule matches the message and (if specified) validate the arguments of the message with regard to a policy violation. If a rule completely matches the invoked Objective-C message, then a policy violation has occurred. Based on the enforcement option we either log the violation, terminate the program, or return shadow data (see also Section 4.1). The same policy enforcement is applied to every API call invoked at runtime, while the difference is that we look up the API call name in the *polHashmap* rather than the class and selector.

While the implementation of the *Log* and *Exit* enforcement option are straightforward (i.e., log the violation in a file, or terminate the program), it is involved to realize the *Replace* option for Objective-C messages. In particular, we achieve this by *replacing* the method implementation at runtime. For this, we request a pointer to the indicated method by contacting the Objective-C runtime. Afterwards, we force the Objective-C runtime to replace the method's implementation with the implementation of a novel and already prepared method that simply returns an empty or arbitrary data structure. Note that we roll back the original method implementation after the shadow data has been returned to ensure a new validation when the same Objective-C message but with a different argument set is invoked.

7. EVALUATION

In this section, we analyze the effectiveness and efficiency of *PSiOS* by using the SpyPhone application [23] as proof-of-concept since it demonstrates which private data can be accessed by every iOS application. Afterwards, we apply *PSiOS* to several popular iOS applications (e.g., Facebook and WhatsApp). Finally, we measure the performance overhead induced by *PSiOS* and compare it to the native runtime execution of applications.

7.1 SpyPhone

SpyPhone is an open-source proof-of-concept application that demonstrates which data can be collected by iOS third-party applications. As SpyPhone is able to retrieve a significant amount of private and sensitive data about the user and the device, we thoroughly analyzed how this is achieved and how the data harvesting can be prevented using *PSiOS*. The data collected by SpyPhone includes, amongst others:

- e-mail account information, including server addresses, user names,
- information on WLANs the device was connected to,
- phone data, such as phone number, UUID, ICCID, IMSI, and call history,
- location data from the weather and the map app,
- history logs from Safari and Youtube,
- personal photos including tagged GPS information,
- address book entries,
- keyboard cache.

[1]Note that the exact section name might differ. However, our implementation does not depend on the section name, instead it uses the `__lazy_symbol` section to locate a `__symbolstub1` like section.

Hence, we have analyzed how SpyPhone collects this data and created policy rules that prevent the requested access. As enforcement type we used the *Replace* option meaning that *PSiOS* returns shadow data for each access request.

Basically, SpyPhone retrieves Wi-Fi configurations, location, call histories, and e-mail account information by accessing property lists that are stored as XML files on any iOS device at the Core OS layer (see Figure 1 in Section 2). For instance, the information about a user's email accounts is retrieved from the file *com.apple.accountsettings.plist*. In general, access to these files can be denied by restricting the Objective-C `dictionaryWithContentsOfFile` method of the `NSDictionary` class, which is used to parse the XML file into an Objective-C data structure. For each file, we defined one rule that restricts the first parameter of the method, namely the filename.

SpyPhone also accesses sensitive information by calling appropriate Objective-C methods at the Core Services Layer (see Figure 1 in Section 2). For instance, this applies to the user's address book or when requesting device information (e.g., the user's phone number). In particular, the address book is used to store private phone numbers, email addresses, and home addresses. We specified policy rules that prohibit SpyPhone from using these Objective-C methods (e.g., `ABAddressBookCopyArrayOfAllPeople` to access the user's address book).

Table 1 shows an excerpt of our policy rules to prevent SpyPhone from accessing private user information. For the sake of completeness the entire sandboxing profile (including all policy rules) for SpyPhone is shown in Appendix B.

7.2 Applying PSiOS to iOS Apps

To demonstrate the effectiveness of our approach, we applied *PSiOS* to a number of popular iOS applications such as Facebook, WhatsApp, ImageCrop, BatteryLife, Flashlight, ImageCrop, InstaGram, MusicDownloader, MyVideo, NewYork (Game), Quickscan, LinPack, Satellite TV and the Audi App. For each app we defined a specific sandboxing profile that prevents the app from accessing private user information. In particular, *PSiOS* successfully prevents access to the address book (for Quickscan, Facebook, and Whatsapp), to personal photos (for ImageCrop and InstaGram), and to the iOS universal unique identifier, short UUID (for Quickscan, BatterLife, Flashlight, MusicDownloader, MyVideo, NewYork, and Audi). As the other applications did not contain any privacy related calls we successfully applied proof of concept enforcement rules to them (access to the file system for LinPack and an URL request for Satellite TV).

7.3 Performance

We conducted a variety of performance tests to assess the practical usability of *PSiOS*. In particular, we performed runtime measurements based on the iOS gensystek benchmark app by using two configurations: (1) *PSiOS* without policy enforcement, but with CFI checks enabled, and (2) *PSiOS* with policy enforcement. For the latter configuration, we applied several policies on Objective-C selectors (such as NSString and NSBundle) which frequently trigger a policy validation for each of the individual benchmarks. The results of our measurements are shown in Figure 4.

Compared to native execution of the individual benchmarks, *PSiOS* (with policy enforcement enabled) only adds

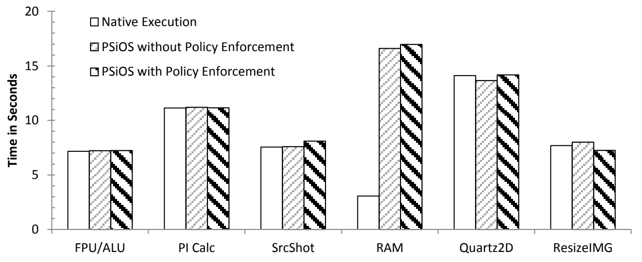

Figure 4: Performance Overhead Measurements with iOS Gensystek App

a negligible overhead ranging from 0.18% to 8%. However, the RAM speed read/write benchmarks induces a significant slowdown of approximately factor 4. Nevertheless, this slowdown is not directly caused by *PSiOS*'s policy enforcement rather by the CFI checks, which are intensively issued for this particular benchmark.

In addition, we measured the overhead of the policy enforcement based on the SpyPhone rule set presented in Section 7.1 using the Instruments application [5]. Figure 5 shows the measured time for native execution, execution with *PSiOS* but without policy enforcement, and execution with *PSiOS* and policy enforcement. Compared to native execution, the load-time overhead of *PSiOS* is only 0.2s. At runtime, the performance overhead for validating an Objective-C message or API call is in the worst-case 0.1s, while sometimes *PSiOS* speeds up the execution. This is due to the fact that we apply the *Replace* enforcement option, which returns shadow data rather than invoking the desired Objective-C call. One of the most called API functions (2163 of 2813 branches and seven percent of the overall execution time) is caused by the Objective-C dispatcher function `objc_msgSend`.

In general, the performance overhead is not noticeable for an end-user, because *PSiOS* enforces access control at the Objective-C layer. Moreover, *PSiOS* applies policy enforcement on the main application binary, while the iOS system libraries and Objective-C frameworks execute without being instrumented by *PSiOS*. Since Apple prohibits developers from using own shared libraries, an adversary cannot deploy own shared libraries to circumvent *PSiOS*.

7.4 Jailbreak Issues

The runtime components of *PSiOS* are completely implemented in one shared library. We opted for this implementation approach, because it enables a system-centric sandboxing solution, and allows every iOS app to immediately benefit from our tools. To install and push our *PSiOS* library on an iOS device, we require a jailbreak of the device, since Apple is closed-source and strictly prohibits any installation of a new shared library. For *PSiOS*, we *only* require a jailbreak for setting a single environment variable, installing a shared library, and allowing our library to rewrite the application code during load-time. Setting the variable as well as installing and signing our library can be easily done by Apple for future iOS releases. Further, for binary rewriting our library only needs to be assigned the dynamic code signing entitlement which allows an application generate code just-in-time.

Information	Policy Rule
Wi-Fi	`<rule type="objc" class="NSDictionary" selector="dictionaryWithContentsOfFile:" mode="replace"> <arg number="1" type="string" operator="=" value="/Library/Preferences/SystemConfiguration/com.apple.wifi.plist"/> </rule>`
Call history	`<rule type="objc" class="NSDictionary" selector="dictionaryWithContentsOfFile:" mode="replace"> <arg number="1" type="string" operator="=" value="/var/mobile/Library/Preferences/com.apple.mobilephone.plist"/> </rule>`
Phone number	`<rule type="objc" class="NSUserDefaults" selector="valueForKey:" mode="replace"> <arg number="1" type="string" operator="=" value="SBFormattedPhoneNumber"/> </rule>`
Address book	`<rule type="api" function="ABAddressBookCopyArrayOfAllPeople" mode="replace" />`

Table 1: An excerpt of the policy rules we applied to SpyPhone

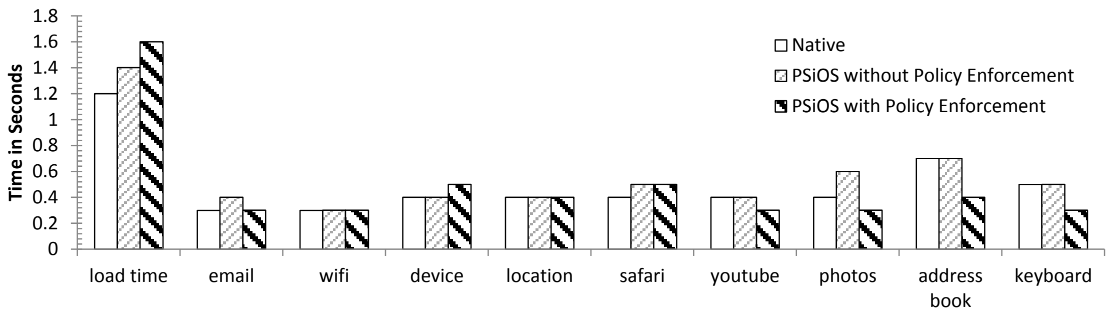

Figure 5: Runtime comparison chart.

Note that many similar approaches targeting Android (for example TaintDroid and AppFence) require the rooting of an Android-powered device as well. Moreover, *PSiOS* does not necessarily require a jailbreak. For instance, *PSiOS* could be provided as a static rewriting tool that Apple applies to all app binaries before releasing them on the App Store. The static analysis (which requires IDA Pro at the moment) could be incorporated using cloud services. Further, app developers could run *PSiOS* before submitting their applications to the App Store. Both approaches are compatible to Apple's signature scheme, since *PSiOS* rewrites the app before it is signed by Apple. We are currently working on the implementation of this approach.

8. RELATED WORK

In this section, we elaborate on related work in the area of mobile security and application sandboxing.

Research on iOS.

The closest work to our framework is PiOS [14] which is a static analysis tool to detect privacy leaks of iOS applications. PiOS generates an application's control-flow graph by backtracking all Objective-C calls and by reconstructing the class inheritance relationships. However, PiOS suffers from some shortcomings: First, its analysis does not cover embedded metaclasses (i.e., root classes of Objective-C classes), which frequently occur in iOS applications. Second, it does not support class clusters (see Section 5). Third, it mainly uses a backtracking of ARM processor registers to determine used classes and selectors. However, as our experiments have shown this approach often fails to resolve the used class. In contrast, *PSiOS* tackles this problem by including the Objective-C sections in its analysis and by retrieving runtime information. To summarize, PiOS is constrained in its analysis because it fails to cover the full picture of the Objective-C runtime, particularly when an adversary deploys obfuscation techniques to circumvent static analysis tools.

Our recent work on iOS has focused on detection of runtime attacks against mobile devices. In particular, our previous work MoCFI [11] enforces control-flow integrity (CFI) on iOS devices running on ARM processors, while CFI [1] has been originally proposed for desktop PCs. However, MoCFI focuses on runtime attacks rather than application sandboxing. Nevertheless, *PSiOS* leverages MoCFI to ensure that our policy enforcement cannot be circumvented by a control-flow attack (see also Section 3).

MobileSubstrate [25] is a framework for jailbroken iOS devices that provides run-time patching of existing programs. To this end, application code can be rewritten to install hooks for Objective-C message handlers and C/C++ functions. The rewriting engine works similar to our approach, however, MobileSubstrate merely provides hooking support. Further, it does not provide protection against runtime attacks which means that a policy framework that is built on top of it could be undermined by runtime attacks.

Research on Android.

In the last years Android has been an appealing subject of research. Kirin [16] is an extended application installer that checks application's permission combinations according to a given policy. Apex [24] goes a step further and allows end-users to choose permissions at install-time.[2] However, since iOS is closed-source, these approaches are not feasible on iOS.

TaintDroid [15] is a framework to detect data leakage attacks on Android. It uses dynamic taint analysis and warns the user whenever sensitive data leaves the device at a taint sink (e.g., the network interface). The AppFence [19] framework builds upon TaintDroid and enables fine-grained privacy rules, and enables the return of shadow data when a policy rule has been violated. However, TaintDroid does not fully cover native code, and could be subverted by runtime attacks. Moreover, it is directly implemented into Android's Java virtual machine (Dalvik). Since iOS use Objective-C rather than the interpreted Java language, it remains open how such a system could be integrated in iOS. Further, in parallel to our work, several security extensions have been proposed to enable fine-grained sandboxing rules (Aurasium [30]) or fine-grained privacy controls [10], but on Android, while we focus on iOS.

Finally, there are several works on Android that tackle privilege escalation attacks at application-level [17, 12, 9]. These attacks are based on the observation that two applications merge their permissions (either directly or indirectly) resulting in a larger sandbox. However, inter-app communication is still an exceptional event on iOS. Nevertheless, in our future work we aim to investigate the feasibility and detection of privilege escalation attacks on iOS with *PSiOS*.

Application Sandboxing Techniques.

Application sandboxing has its origins in the field of software fault isolation (SFI) [29]. The basic idea of SFI is to isolate code and data of an untrusted module in a separate fault domain. Afterwards, the untrusted module is instrumented to ensure that the code cannot jump or reference data beyond its fault domain. In particular, NativeClient [31, 27] builds upon SFI and enables a sandboxed environment for native code plugins in web browsers. Our iOS policy framework is closely related to these works. In particular, we enable SFI for iOS applications, by instrumenting all calls to the Objective-C runtime. However, note that the existing works either focus on entirely different computing platforms or are incompatible to Objective-C.

9. CONCLUSION

In this paper, we introduced *PSiOS*, a novel policy enforcement framework for the closed-source mobile operating system iOS. *PSiOS* provides fine-grained, application-specific, and user-driven sandboxing for third-party applications *without* requiring access to source code. We presented the design decisions behind the framework and discussed in detail the improvements to prior work in this area. We implemented a fully working prototype of *PSiOS* and also illustrated technical details of the implementation. In an empirical evaluation, we demonstrated that *PSiOS* effectively prevents privacy breaches by testing our reference implementation with *SpyPhone* [23], a tool specifically designed to steal data from an iOS device. The runtime overhead introduced by our tool is reasonable and we demonstrated that even complex applications can be instrumented by *PSiOS*. In our future work, we aim to provide *PSiOS* as a static rewriter that can be leveraged by app developers and App Store manufactures to allow fine-grained application sandboxing without requiring users to jailbreak their device.

[2]Note that the standard Android system follows the all-or-nothing principle.

10. ACKNOWLEDGEMENTS

This work has been supported by the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (Grant IV.5-43-02/2-005-WFBO-009).

11. REFERENCES

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

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

[3] Apple Inc. Mac OS X ABI Mach-O File Format Reference. `http://developer.apple.com/library/mac/documentation/developertools/conceptual/MachORuntime/Mach-O\_File\_Format.pdf`, 2009.

[4] Apple Inc. Objective-C Runtime Programming Guide. `http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/ObjCRuntimeGuide.pdf`, 2009.

[5] Apple Inc. Instruments user guide. `https://developer.apple.com/library/mac/\#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html`, 2010.

[6] Apple Inc. Entitlement key reference. `http://developer.apple.com/library/mac/#documentation/Miscellaneous/Reference/EntitlementKeyReference/Chapters/EnablingAppSandbox.html#//apple_ref/doc/uid/TP40011195-CH4-SW1`, 2011.

[7] Apple Inc. App Store Review Guidelines. `https://developer.apple.com/appstore/guidelines.html`, 2012.

[8] Apple Inc. ios sdk release notes for ios 6. `http://developer.apple.com/library/ios/#releasenotes/General/RN-iOSSDK-6_0/index.html`, 2013.

[9] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Symposium on Network and Distributed System Security (NDSS)*, Feb 2012.

[10] S. Bugiel, S. Heuser, and A.-R. Sadeghi. myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android. Technical Report TUD-CS-2012-0226, Center for Advanced Security Research Darmstadt (CASED), 2012.

[11] 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.

[12] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight provenance for smartphone operating systems. In *USENIX Security Symposium*, 2011.

[13] J. Duart. Objective-C helper script. https://github.com/zynamics/objc-helper-plugin-ida.

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

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[16] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[17] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[18] Google. Security and permissions. http://developer.android.com/guide/topics/security/security.html, 2012.

[19] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the Droids you're looking for: Retrofitting Android to protect data from imperious applications. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2011.

[20] V. Iozzo and R.-P. Weinmann. Pwn2Own contest. http://blog.zynamics.com/2010/03/24/ralf-philipp-weinmann-vincenzo-iozzo-own-the-iphone-at-pwn2own/, 2010.

[21] J. Lowensohn. US Congress probes iOS app makers over privacy. http://bit.ly/IuqIFu, March 2012.

[22] C. Miller and D. Blazakis. Pwn2Own contest. http://www.ditii.com/2011/03/10/pwn2own-iphone-4-running-ios-4-2-1-successfully-hacked/, 2011.

[23] N. Seriot. SpyPhone. https://github.com/nst/SpyPhone, 2011.

[24] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[25] saurik. MobileSubstrate. http://iphonedevwiki.net/index.php/MobileSubstrate.

[26] M. J. Schwartz. iOS Social Apps Leak Contact Data. http://www.informationweek.com/news/security/privacy/232600490, 2012.

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

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

[29] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5), 1993.

[30] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security Symposium*, 2012.

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

APPENDIX

A. LOAD COMMANDS

Our static Objective-C analyzer parses every load command included in the Mach-O file header and in particular inspects the load commands listed in Table 2.

Load command	Purpose
`LC_SEGMENT`	Defines code and data segments, while each segment contains of a set of sections
`LC_SYMTAB`	Specifies the symbol table
`LC_DYLD_INFO` and `LC_DYLD_INFO_ONLY`	Define binding information

Table 2: Relevant Mach-O Load Commands

B. SANDBOXING PROFILE FOR SPYPHONE

```
<rule type="api" function="
      ABAddressBookCopyArrayOfAllPeople:" mode="
      replace" />
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/Library/Preferences/SystemConfiguration/
       com.apple.wifi.plist"/>
</rule>
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Preferences/com.apple.
       accountsettings.plist"/>
</rule>
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Preferences/com.apple.
       accountsettings.plist"/>
</rule>
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Preferences/com.apple.
       commcenter.plist"/>
</rule>
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Preferences/com.apple.
       mobilephone.settings.plist"/>
</rule>
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Preferences/com.apple.
       mobilephone.plist"/>
</rule>
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Preferences/com.apple.
       Maps.plist"/>
</rule>
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Preferences/com.apple.
       preferences.datetime.plist"/>
</rule>
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Preferences/com.apple.
       weather.plist"/>
</rule>
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Preferences/com.apple.
       mobilesafari.plist"/>
</rule>
<rule type="objc" class="NSDictionary"
     selector="dictionaryWithContentsOfFile:" mode=
          "replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Preferences/com.apple.
       youtube.plist"/>
</rule>
<rule type="objc" class="UIDevice" selector="
      uniqueIdentifier:"
     mode="replace" />
<rule type="objc" class="NSUserDefaults" selector=
      "valueForKey:"
     mode="replace">
 <arg number="1" type="string" operator="=" value=
       "SBFormattedPhoneNumber"/>
</rule>
<rule type="objc" class="NSFileManager" selector="
      directoryContentsAtPath:" mode="replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Library/Keyboard/"/>
</rule>
<rule type="objc" class="NSFileManager" selector="
      enumeratorAtPath:"
     mode="replace">
 <arg number="1" type="string" operator="=" value=
       "/var/mobile/Media/DCIM"/>
</rule>
```

Listing 1: Full rule set for SpyPhone