XiOS: Extended Application Sandboxing on iOS

Mihai Bucicoiu[1], Lucas Davi[2], Razvan Deaconescu[1], Ahmad-Reza Sadeghi[2]

[1] University "POLITEHNICA" of Bucharest, Romania
{mihai.bucicoiu,razvan.deaconescu}@cs.pub.ro

[2] Technische Universität Darmstadt
Intel Collaborative Research Institute for Secure Computing, Darmstadt, Germany
{lucas.davi,ahmad.sadeghi}@trust.cased.de

ABSTRACT

Until very recently it was widely believed that iOS malware is effectively blocked by Apple's vetting process and application sandboxing. However, the newly presented severe malicious app attacks (e.g., Jekyll) succeeded to undermine these protection measures and steal private data, post Twitter messages, send SMS, and make phone calls. Currently, no effective defenses against these attacks are known for iOS.

The main goal of this paper is to systematically analyze the recent attacks against iOS sandboxing and provide a practical security framework for iOS app hardening which is fully independent of the Apple's vetting process and particularly benefits enterprises to protect employees' iOS devices. The contribution of this paper is twofold: First, we show a new and generalized attack that significantly reduces the complexity of the recent attacks against iOS sandboxing. Second, we present the design and implementation of a novel and efficient iOS app hardening service, *XiOS*, that enables fine-grained application sandboxing, and mitigates the existing as well as our new attacks. In contrast to previous work in this domain (on iOS security), our approach does *not* require to *jailbreak* the device. We demonstrate the efficiency and effectiveness of *XiOS* by conducting several benchmarks as well as fine-grained policy enforcement on real-world iOS applications.

Categories and Subject Descriptors

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

Keywords

binary instrumentation; sandboxing; mobile security; iOS

1. INTRODUCTION

iOS is after Android the most popular mobile operating system worldwide. It is deployed on well-known Apple devices such as iPhone, iPad, or iPod Touch, used by millions of users everyday. Apple maintains an app store, first introduced in July 2008, that hosts in Sep. 2014 more than 1,300,000 applications (apps) [28].

On the other hand, the popularity, the high number of features and apps, as well as the large amount of sensitive and private information that are available on iOS devices make them attractive targets for attackers. To address the security and privacy concerns, iOS enforces two main security principles: code signing and application sandboxing. The former ensures that only Apple-approved software can be executed on an iOS device while the latter technique guarantees that an app only performs operations within the boundaries of a pre-defined sandbox, and can neither disrupt nor access other applications. In particular, iOS distinguishes between *public* and *private* frameworks in its sandboxing model. Public frameworks comprise shared system libraries and APIs that can be accessed by *every* third-party app. For instance the AddressBook framework is a well-known available public framework that a developer is allowed to use in an iOS application [6]. In contrast, private frameworks and APIs should only be accessed by system applications. Prominent examples for private APIs are sending SMS messages or setting up a call.

As another line of defense, Apple also conducts application review/vetting for all apps that are to be published on the App Store, and rejects any application that attempts to invoke a private API. The folklore belief is that the vetting process is sufficient to effectively block malware from entering the App Store [17, 32].

However, recent research results shed new light on the security of iOS by demonstrating how to bypass the vetting process and iOS application sandboxing [19, 35]. The main idea of these attacks is to dynamically load private frameworks and invoke private APIs to induce malicious behavior without the user's consent. The attacks range from sending SMSs or emails to attacker specified addresses, posting Twitter tweets, abusing camera and audio, setting up phone calls, and stealing the device ID.

Some of these attacks use return-oriented programming [30] to hide and obfuscate the malicious functionality [35]. Such techniques impede the detection of malicious code and manifest the limits of any *off-line* application vetting approach.

Preventing these attacks is highly challenging due to the current system design of iOS. Prohibiting applications to load private frameworks is not a solution since public frameworks are not self-contained and need to interact with private frameworks to complete their tasks. Moreover, some public frameworks also contain private (hidden) API func-

ASIA CCS '15, April 14 - 17, 2015, Singapore, Singapore
Copyright © 2015 ACM 978-1-4503-3245-3/15/04 ...$15.00.
http://dx.doi.org/10.1145/2714576.2714629.

tions, e.g., sending a Twitter message in background is a private API call inside a public framework [19].

Moreover, the recently proposed security enhancements for iOS (see Section 7) suffer from various deficiencies: these are either static and cannot detect attacks that trigger malicious behavior (only) at runtime [16], or require a jailbreak and induce performance overhead [15, 37].

Our goal in this paper is to provide a framework for app hardening that enables fine-grained app sandboxing on iOS and tackles the shortcomings of existing solutions, i.e., defeating the recent attacks against iOS sandboxing [19, 35]. In particular, we make the following contributions:

New malicious app attacks. We investigate previous attacks against iOS application sandboxing and significantly reduce their complexity. We show a new attack that requires no specific use of a public framework to access a private API, but rather exploits the *default* memory layout used in *all* iOS applications to invoke any private API of the adversary's choice. This allows us to construct general attack vectors on iOS application sandboxing.

Defense tool. We present the design and implementation of a novel mitigation service and tool, termed *XiOS*, for fine-grained application sandboxing on iOS tackling these attacks at API and function-level. In contrast to previous work in this domain [15, 37], our solution performs efficiently without requiring a *jailbreak* of the underlying device. We developed a new static binary rewriter for iOS that implants a reference monitor into the application's code, and protects, under reasonable assumptions, the reference monitor from the (potentially malicious) application without requiring the *source code* of the application. Our defense mechanism instruments all API calls and provides the following features: (i) API address hiding, and (ii) optional policy checks based on user, developer or administrators (e.g., enterprises) defined policies for public API calls.

Evaluation. We show the effectiveness and efficiency of our approach by applying it to several existing real-world iOS applications including Gensystek, WhatsApp, System Monitor, Music Notes and Frotz. Our evaluation shows that no overhead is added in terms of user-experience when *XiOS* is used to protect the application. We also show fine-grained policy enforcement for WhatsApp, allowing filtering of contacts information (see Section 6.4).

Advantages of our solution. Our defense technique does neither require a *jailbreak* nor the application's *source code.* Since our defense is implemented as a static binary rewriter, it can be applied just before an application is submitted to the App Store. On the one hand, it provides a useful tool for benign developers to harden their app from being compromised by a remote exploit that invokes private APIs. On the other hand, it can also be used by Apple to improve the vetting process by hardening potentially malicious apps with *XiOS*. In other words, *XiOS* (1) constrains malicious apps in their actions, and (2) hardens benign apps.

Compliance and independency. Our solution adheres to the existing design of private and public frameworks in iOS. It allows for flexible policy enforcement giving developers the possibility to define fine-grained access control policies for each app according to their own requirements. Finally, our solution is not dependent on application vetting, and can be deployed by enterprises to protect employees' devices, and still allowing the employees to use popular apps such as WhatsApp (see Section 4.5, and 6.3).

2. BACKGROUND

In this section, we recall the iOS security architecture and elaborate on related attacks against app sandboxing.

2.1 iOS Security

The main security mechanisms used on iOS are (1) code signing, (2) application vetting, (3) file system and data encryption, (4) memory randomization (ASLR) along with non-executable memory, and (5) application sandboxing.

Code signing ensures that only Apple-signed software can be executed on an iOS device. To bypass this restriction, users can jailbreak (root) their devices which allow them to arbitrarily install non-approved Apple software. Apple approves signed applications after a vetting process. Although the implementation details of application vetting are not public, Apple states that it *reviews all apps to ensure they are reliable, perform as expected, and are free of offensive material* [4]. Apple also deploys an AES-256 hardware crypto engine to encrypt the file system of an iOS device.

Address space layout randomization (ASLR) randomizes the start addresses of data and code segments. This makes runtime attacks like return-oriented programming that rely on fixed code addresses more cumbersome. Typically, ASLR is combined with the non-executable memory security model, which enforces that a memory page cannot be writable and executable at the same time [25]. This technique prevents runtime attacks that attempt to inject malicious code into an application's address space and execute it. iOS goes even one step further, and enforces code signing on memory pages at runtime: it prohibits any third-party app from dynamically generating code or changing existing (mapped) code.

An abstract view of the security architecture to realize *application sandboxing* on iOS is shown in Figure 1. iOS deploys sandboxing to isolate applications from each other, and to control access of applications to the operating system. In particular, we distinguish components on three software layers: (1) the kernel layer which provides basic system services (file system and network) and a kernel module to realize application sandboxing, (2) the Objective-C framework layer and a privacy setting service, and (3) the application layer where third-party and built-in apps are executing.

The main component to enforce application sandboxing resides in the iOS kernel, namely a TrustedBSD mandatory access control (MAC) module. This kernel module enforces sandboxing at the level of system calls and directory paths. Further, sandboxing is driven by sandboxing profiles which are pre-defined by Apple. The profiles consist of access control lists (ACLs) that either deny or grant access to certain system calls and file paths.

Apple defines a single sandboxing profile for third-party apps. Hence, all apps execute with the same privilege level. In particular, this profile prohibits App A to access code or data from other applications like App B (see Figure 1).

Apart from the TrustedBSD kernel module, there are several restrictions imposed by Apple indirectly related to application sandboxing. As mentioned before Apple distinguishes between public and private frameworks[1]. Private frameworks are reserved for iOS built-in and system applications. Although third-party applications are only allowed to access public APIs of a public framework, there is no fun-

[1] The list of all available frameworks can be downloaded from `http://theiphonewiki.com/wiki/System/Library/Frameworks`

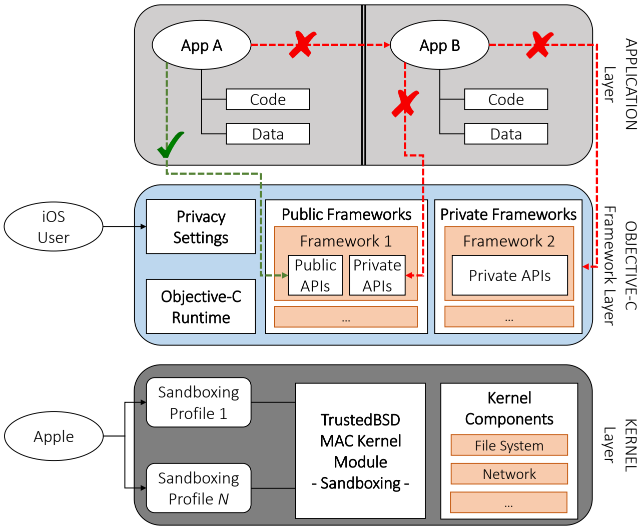

Figure 1: Basic iOS architecture to enforce application sandboxing

damental operating system mechanism that prevents the use of private APIs. Instead, Apple relies on the application vetting process to discover such unauthorized access requests.

Finally, since iOS version 6, iOS allows users to specify privacy settings on a per-app basis. Typically, iOS apps have by default access to private information such as contacts, device IDs, keyboard cache, or location. In order to restrict the access to this information, iOS users can arbitrarily configure privacy settings. In fact, this allows users to specify restrictions on some selected privacy-related *public* APIs. However, there is no general for all non-privacy related public APIs as well as private APIs.

2.2 Related Attacks on iOS Sandboxing

Recent attacks deploy a malicious third-party application that dynamically loads private frameworks and invokes private APIs without being detected by Apple's vetting process [35, 19]. In order to understand these attacks, we need to take a deeper look at how legitimate calls to private APIs in system apps are handled.

Typically, a call to a private API is internally handled as an external function call to a shared library. For this, a program requires that the library encapsulating that function is loaded into the application's address space. In addition, the runtime address of the desired function needs to be populated. In practice, this is achieved by dynamic loading [24]. In iOS, dynamic loading is provided by a dedicated library called `libdl.dylib`. We refer to this library as the dynamic loader. Specifically, the dynamic loader provides two fundamental methods: (1) *load-library* via the function `dlopen`, and (2) *load-address* using the `dlsym` function that determines the runtime address of a function residing in a library that has been already loaded. If an executable module (i.e., an application or a shared library) attempts to invoke a private API, then the linker will add the corresponding placeholders into the data section of the executable module. These placeholders will be automatically initialized with the correct addresses of *load-library* and *load-address* at load-time. Once the executable module starts executing, it can invoke *load-library* to dynamically open a private framework, and subsequently issue *load-address* to retrieve the address of a private API and call it.

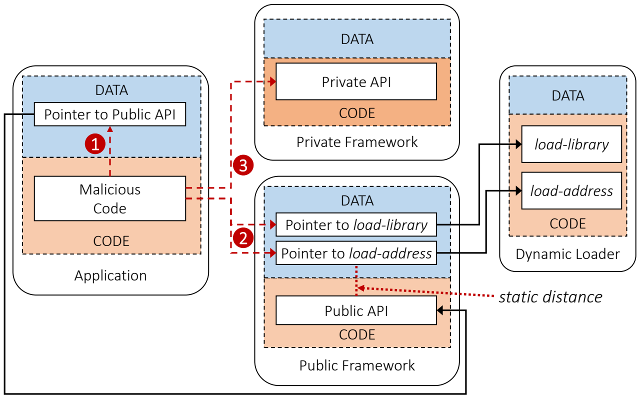

Figure 2: Dynamically invoking private APIs

However, Apple prohibits any third-party application from using *load-library* and *load-address*, thereby preventing attempts to invoke private APIs. In the vetting process, one can simply check whether the application contains placeholders for these functions in the data section. On the other hand, public frameworks (developed by Apple) are allowed to use these functions. This leads to a confused deputy problem [20]: since third-party applications are allowed to load public frameworks, they can potentially misuse them to call a private API.

Figure 2 shows the general workflow of these attacks in detail. The malicious application legitimately links to a public framework (that uses *load-library* and *load-address*) and invokes a public API for benign reasons. Hence, the dynamic iOS loading process will populate the address of the public API into the data section of the malicious application. This allows the adversary to read out the address of the public API at runtime by de-referencing the designated placeholder in the data section (step 1). Effectively, this reveals the runtime location of the public API in memory. Based on this runtime address the adversary can determine the location of the two placeholders where the runtime addresses of *load-library* and *load-address* are stored (step 2). This is possible, because the relative offset (i.e., distance) between the start of the public API function and the two placeholders is constant, and can be pre-computed prior to execution. Once the adversary knows the address of *load-library* and *load-address*, he can call these functions to dynamically load a private framework of his choice and execute a private API (step 3).

In summary, existing attacks against iOS application sandboxing require (i) the availability of a public framework that necessarily uses dynamic loader functions *load-library* and *load-address*, and (ii) two address de-reference operations: one for the public API and another one for the address of either *load-library* or *load-address*.

3. DEVELOPING IMPROVED ATTACKS

Recent attacks require a public framework that either invokes *load- library* or *load-address* (as described in Section 2.2). Our investigation showed that out of 319 available public frameworks only 25 use *load-library* and 36 *load-*

address. Hence, one theoretical approach to defend against this attack is to rewrite some of the public frameworks and disable dynamic loading for them. However, by systematically analyzing and reverse-engineering the lazy binding mechanism used in iOS, we were able to launch the same attacks without requiring the application to link to *any* of the public frameworks. This not only significantly reduces the complexity of the attacks but also allows more general attacks, because the mechanisms we exploit in our attacks are by default enabled in *every* iOS application.

Specifically, we developed a simple iOS application using the standard settings in Xcode (which is the main IDE for iOS app development). By default, our application deploys lazy binding mechanisms. We traced back at assembler and binary-level[2] how lazy binding is performed in iOS, and were able to recognize that the lazy binding implementation is vulnerable to even simpler attacks.

In general, lazy binding dynamically (and transparently to the application) resolves the runtime address of a symbol (e.g., an external function or variable) the first time it is used [2]. In contrast, non-lazy binding resolves all the addresses of symbols once at application load-time. To support lazy binding, iOS maintains two data sections in iOS applications, one for non-lazy symbols and one for lazy symbols. One important symbol that is *always* defined in the section for non-lazy symbols is the external function `dyld_stub_binder`. This particular function realizes lazy binding: it resolves the runtime address of an external function that is defined in the lazy symbol section when it is accessed for the first time. Remarkably, this function is part of the dynamic loader library, the same library that contains *load-library* and *load-address*. Hence, an adversary can directly infer the address of the dynamic loader functions by de-referencing the address of `dyld_stub_binder` from the data section of the malicious application. Our improved attack completely removes the operations and attack requirements for Step 2 in Figure 2.

Note that the de-referencing of `dyld_stub_binder` makes no use of the string `"dyld_stub_binder"`, making the attack vector stealthy to static analysis methods. Its location offset inside the executable is statically precomputed using binary analysis. A potential attack would use the computed offset to retrieve the address of `dyld_stub_binder` from its location and then compute the address of *load-library* or *load-address*. This can be achieved offline due to the fact that all three functions are stored within the same library.

In contrast to previous work on bypassing iOS sandboxing [19, 35], our new attack does not require any specific public frameworks to be loaded. We also reduce the necessary attack steps: we only require a single address read from the data section of the application. Our attack has important implications: it can be applied to any iOS application since lazy binding is currently used for all iOS applications. Hence, given a benign application that contains a vulnerability, we can arbitrarily invoke private APIs using return-oriented programming.

In order to show the effectiveness of our new attack, we have developed an application for iOS 7 that dynamically invokes private APIs at runtime. We added to the application malicious code that can be used at runtime to derive the address of the private *createScreenIOSurface* API from the *UIWindow* class. Similar to previous attacks, our attack is triggered remotely, and continuously takes snapshots and sends them to a remote server controlled by us. We obfuscated and submitted our application to the App Store, and successfully passed the vetting process. Hence, our malicious app can still invoke private APIs in a stealthy manner without being detected by the vetting process using less de-referencing actions than existing attacks. The application was installed only on our testing device and then removed from the App Store. Moreover, we also stopped the server which remotely triggered the attack.

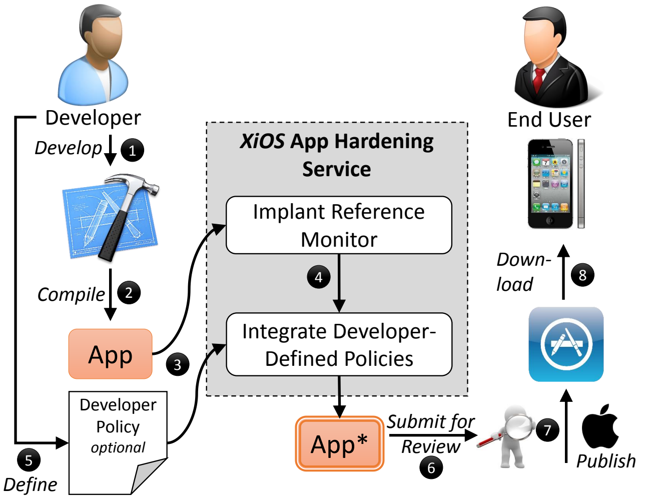
Developer
Develop 1
Compile 2
App
3
Developer Policy optional
5 Define
XiOS App Hardening Service
Implant Reference Monitor
4
Integrate Developer-Defined Policies
App*
Submit for Review 6
7
Publish
End User
Down-load 8

Figure 3: General approach and workflow of *XiOS*

4. OUR DEFENSE SERVICE *XiOS*

Mitigation of attacks against iOS sandboxing and, in particular our new attacks, is a challenging task due to several reasons: First, the attacks are based on exploiting lazy binding which is enabled by default on iOS due to performance reasons and cannot be simply disabled. Second, public frameworks are tightly coupled to private frameworks. Removing this interdependency would induce heavy design changes. Third, iOS is closed-source preventing direct extensions of the operating system with a reference monitor for private APIs.

In this section, we introduce the requirements on *XiOS* describe its workflow and system model, its components, and the corresponding security aspects. Finally, we discuss real-world deployment scenarios for *XiOS*.

4.1 Requirements and Assumptions

Given the above mentioned challenges, a defense tool needs to meet the following functional and security requirements:

R1: Preserve the benefits and efficiency features. In particular, mechanisms such as lazy binding should further be in place.

R2: Require no changes to the operating system and current software stack architecture of iOS. In particular, we need to ensure that private APIs are still accessible from public frameworks.

R3: Require no application's source code. Typically, the source code of iOS applications is not available, even

[2] using the Xcode debugger and IDAPro as disassembler

Apple's App Store only retrieves an application as a binary bundle.

R4: Integrate seamlessly into the existing application development and distribution process of iOS. In particular no jailbreak of the user's device is needed.

S1: Prevent (previous) attacks on iOS sandboxing [19, 35], and our novel attack (Section 3).

S2: The malicious application cannot bypass or disable our reference monitor (see Section 4.4).

We build our defense on the following assumptions:

A1: We assume that the target system enforces the principle of non-executable memory. Otherwise, an adversary could revert our security checks and circumvent our defense. As mentioned in Section 2.1, iOS deploys a very strict version of non-executable memory.

A2: Since our defense operates at application-level (due to R2), we obviously assume the underlying operating system kernel to be benign and not compromised.

A3: An adversary is not able to statically pre-compute the address of a private API or a critical function such as *load-library* and *load-address*. This is ensured by iOS since it applies ASLR to shared libraries.

4.2 System Model

As mentioned above (S1), the main goal of our solution is to prohibit the invocation of any private API from a third-party app to prevent attacks against app sandboxing. At the same time, we need to ensure that public frameworks can still access private APIs. Moreover, we generalize our design to enable fine-grained access control rules on public APIs. Our new hardening service *XiOS* achieves both (i) preventing invocations of private APIs, and (ii) enforcing developer-defined fine-grained access control rules on public APIs.

The workflow of *XiOS* is depicted in Figure 3. Our hardening service seamlessly integrates into Apple's application development process: after the developer has finished programming the app and compiling it (Step 1 and 2), the application is submitted to our hardening service via a web browser (Step 3). Next, the hardening process implants the *XiOS* reference monitor and some additional startup code into the application. The reference monitor and the startup code will hide traces of external API addresses including the address of `dyld_stub_binder`. Moreover, if the developer appended a policy for public APIs, we also embed these policies into the application (Step 4). To accomplish these tasks, *XiOS* makes use of binary instrumentation techniques (R3). Finally, the app is submitted to the App Store and can be later installed on users' devices (Step 6 to 8).

Our service always implants the reference monitor to prevent invocation of private APIs through the dynamic loader. In addition, it allows the developer to *optionally* append a custom policy for public APIs (Step 5). For instance, one could define a policy that restricts a messenger app (e.g., WhatsApp) to only upload a subset of the address book to the app server (due to privacy reasons). For this, we support different policy enforcement options: `allow`, `deny`, `log` or `modify`. In particular, the `modify` option allows the replacement of the arguments passed to an external function and public API return values. Defining such policies is tightly coupled to the application purpose and its target users. Enterprises can integrate *XiOS* within their mobile device management (MDM) systems and harden all applications before they are deployed on employees' (personal) devices. Hence, enforcement of privacy-related policies can protect a company from being legally held responsible for information leakage [33].

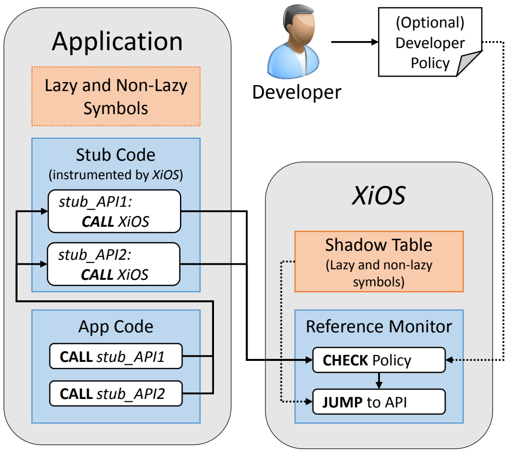

Figure 4: Instrumented *XiOS* App

It is also possible that an end-user submits an existing app to our service along with policies she desires to be enforced on that app. However, post-changes on existing apps would change the hash value of the application. This would imply re-submission to Apple's App Store to retrieve an updated Apple certification. Hence, the main purpose of *XiOS* is app hardening for both vulnerable benign and potentially malicious apps before Apple signs and uploads the app on the App Store.

4.3 Architecture

The high-level idea of *XiOS* app instrumentation is depicted in Figure 4. *XiOS* is directly implanted into the original iOS application (R2 and R4) and mainly contains a reference monitor that mediates all access requests of an application to an external library. The idea is here to completely hide public API calls and the `dyld_stub_binder` address from the app and redirect these requests to the reference monitor which is the only component that knows and can retrieve the runtime addresses of external functions.

There are three binary sections which are of particular interest: (1) the main app code section, (2) the so-called symbol stub code section, and (3) a data section where the lazy and non-lazy symbol pointers are stored. These sections correspond to the following iOS Mach-O binary sections: `__text` for the main app code, `__symbol_stub` for the symbol code section, `__la_symbol_table` and `__nl_symbol_table` for the lazy and non-lazy symbol data section.

Typically, when an iOS application attempts to invoke an external library function (i.e., a public API), the compiler will insert a dedicated stub for the desired function in the code stub section. Hence, the originating call to that external function effectively redirects the control-flow to the stub code section, e.g., a call to *stub_API1* or *stub_API2* in Figure 4. The stub code reads out the runtime address of the external function in either the lazy or non-lazy symbol data section. When the runtime address of the external function is not yet known, the stub code will ask the dynamic loader to resolve the runtime address and eventually invoke the external function.

To prevent exploitation of the lazy binding process and the leakage of important runtime addresses, we instrument the stub code section and remove its connection to the lazy symbol section. To this end, *XiOS* creates a duplicate of the lazy and non-lazy symbol section in a dedicated memory area denoted as *shadow table*. It also overwrites the original section with zeroes. In addition, we instrument each stub entry so that it always invokes our *XiOS* reference monitor when an external function is called. Thus, we ensure that all calls to external functions are redirected to our reference monitor, and simultaneously guarantee that a malicious application cannot read runtime addresses from the lazy and non-lazy symbol sections to launch attacks against app sandboxing.

Upon invocation, the reference monitor of *XiOS* performs as follows: first, it determines which external function the application attempts to execute. This is achieved by looking up the corresponding entry in the duplicate of the lazy and non-lazy symbol section. Next, it performs a policy check. For the invocation of public APIs, the (optional) developer-defined policies are consulted. The invocation of private APIs is prevented by the design of our reference monitor as it hides the crucial runtime addresses of public APIs and dynamic loader functions. However, the challenge is to still preserve the performance benefits of lazy binding. We tackle this as follows: when an external function is called for the first time, our reference monitor asks the dynamic loader to resolve the runtime address, and stores its value for later invocations in the shadow table.

4.4 Security Considerations

As mentioned in Section 4.1, we need to ensure the integrity of *XiOS* and prevent the untrusted application from reading the shadow table (S1 and S2). This is challenging as *XiOS* resides in the application's address space.

As explained above, the shadow table contains runtime addresses of public APIs and the dynamic loader functions. If an adversary can access these addresses then the attacks described in Section 2.2 and 3 can be launched. Ideally, the *XiOS* reference monitor should be executed in a separate process. However, due to the closed iOS, we can only implant it into the application itself to avoid a jailbreak (R4).

To only allow the reference monitor to access the shadow table, we need to enforce software fault isolation (SFI) [34], i.e., basically dividing an application in trusted and untrusted code parts. Since realizing SFI is cumbersome [40], we opted for a pragmatic SFI solution for our proof-of-concept implementation: Whenever the (untrusted) main application code is executing, *XiOS* marks all memory pages allocated for the shadow table as non-readable. In contrast, when the control is transferred to the reference monitor, *XiOS* changes the access rights of the pages back to readable. We demonstrate in Section 6 that modifying the access rights of the memory pages performs efficiently in *XiOS*. Although we fully support multithreading, an adversary can potentially trigger a policy validation, and force the iOS scheduler to switch to another thread while the shadow table is readable. We have not addressed this issue in our current proof-of-concept implementation, but currently we are working on ensuring that no other program thread is allowed to execute while the shadow table is set to readable.

As *XiOS* invokes memory management functions to prevent the main application from accessing the shadow table, we need to prevent the main application to exploit system calls to get access to the shadow table. Although it might seem that applications become limited by not allowing them to directly invoke system calls, this is not the case for iOS. Note that applications can still indirectly use system calls through external API functions that wrap their functionality and are mediated by our reference monitor. We prohibit all applications to directly invoke system calls using an in-house developed script that identifies invocations of system calls. Moreover, we have analyzed a large number of popular iOS applications (e.g., Facebook, Google Maps, YouTube, Angry Birds, Skype) and noticed that none uses directly system calls, hence, *XiOS* can be applied to them.

We also need to ensure that at application-start *XiOS* takes over the control first. Otherwise, an application could access the non-lazy symbol section to retrieve critical runtime addresses. Hence, we dispatch the entry point of an application such that *XiOS* is always executed first to set-up the shadow table and zeroing out the original lazy and non-lazy symbol sections.

XiOS prevents attacks that attempt to jump over policy validation code. Since we instrument all stubs (see Figure 4), we ensure that policy validation always starts from its original entry point. Note that stub code cannot be modified by an adversary as the code is mapped as non-writable. A last chance for the adversary is to exploit an indirect jump, call or return to redirect the control-flow to the place after the policy check. These attacks can be prevented by control-flow integrity (CFI) for iOS [15], or by computing a secret value at the beginning of the policy check and checking the secret value at return of *XiOS* (see for instance checkpoint handling in [27]). We leave this as future work to further extend and improve *XiOS*.

4.5 XiOS Deployment

XiOS can be mainly deployed as an off-line remote hardening service in two scenarios. Primarily, it can be deployed by Apple or an enterprise before a new app is uploaded to the application store. This provides protection against malicious apps such as the one described in [19, 35])3. Alternatively, it can be deployed by (benign) developers that aim at hardening their apps from being remotely exploited and enforcing access control rules on public APIs.

For the former scenario, an enterprise can integrate *XiOS* into its Mobile Device Management infrastructure. Apple actively supports enterprise-specific app stores where employees can run applications that only need to be signed by the developer and the enterprise [5]. In this case, the applications are instrumented by *XiOS*, signed by the enterprise's administration, and the employee can download and install the hardened application over a custom enterprise application store.

The latter scenario allows benign developers to deploy *XiOS* to enforce fine-grained access control rules on public APIs. This provides better security than directly integrating access control rules into an app's source code, because an adversary could bypass these checks by jumping over the policy validation instructions. We prevent such attacks by enforcing every external function call to be dispatched over our instrumented code stubs. We are also currently working on an *XiOS* extension where such policies can be downloaded from and defined on a remote server.

5. IMPLEMENTATION

In this section we detail on our current implementation of *XiOS*. Considering that our hardening process works directly on the application's binary executable, and that the Mach-O file format is used by *all* iOS applications regardless of the operating system version, our implementation is applicable to any existing iOS app.

The first step of the hardening process is to search for direct invocations of system calls. To do so, we use binary disassembly tools (specifically IDAPro) to identify the assembler instruction used for system calls. On ARM, one can directly invoke a system call by using the `svc` (supervisor call) instruction[3]. In contrast to the x86 architecture, ARM instructions are aligned and any unaligned memory access generates a fault exception. Hence, an adversary must explicitly insert the `svc` instruction. We search for these instructions and raise an alarm if an app is using such an instruction.

In the remainder of this section we describe how calls to external functions are dispatched by *XiOS* and present the implementation of our reference monitor.

5.1 Dispatching External Function Calls

As already mentioned in Section 4.3, we instrument the stub code of an application to redirect execution to our reference monitor. Originally, each entry i in the stub code section consists of a load instruction that loads the address placed at entry i in the lazy symbol section into the program counter `pc`. Effectively, this instruction realizes an indirect jump where the target address is taken from memory (i.e., from the lazy symbol section). Recall that this requires the dynamic loader to relocate the runtime addresses of external functions in the lazy symbol section section either at load-time or on-demand at runtime, c.f. Section 2.

In our implementation, we replace the `pc`-load instruction in the stub code with a branch instruction that targets our reference monitor. Since both instructions have the same size no realignment of the binary is required. Specifically, we overwrite the original `ldr` (load register) instruction with a `b` (branch) as follows:

```
ldr pc,[lazy_sym_i] -> b reference_monitor
```

5.2 Reference Monitor

The reference monitor achieves two important goals: (1) replaces the iOS dynamic loader and offers a mechanism for dynamically searching the addresses of external functions while protecting them from being leaked to the main application, and (2) enforce developer-based policies.

The pseudo code of the reference monitor is shown in Algorithm 1. First, we determine the name and parameters of the hooked functions (lines 1-2). Second, based on the enforcement rules defined in the hardening process, we determine if the call to the external function is allowed or blocked (line 3). If the access to the external function is not granted, the program exits and, thus, blocks any information leakage. Third, if invocation is allowed and after marking the shadow table as readable and writable (line 5), the reference monitor verifies if a previous call to the same function has been made (line 6). If so, the address of the function can be retrieved from the shadow table (line 7) and the control-flow can be transferred to the external function (line 12). In case the function is called for the first time, we make use of the standard iOS dynamic loader to resolve the runtime address of the external function (line 9) and store it into the shadow table for future invocations (line 10). Finally, the reference monitor marks the shadow table as non-readable and non-writable (line 13).

Algorithm 1 Reference monitor - pseudo-code

```
 1: params[] = read_params_from_registers();
 2: fnct_name = decode_function_name();
 3: execution = apply_pre_policies(fnct_name, params[]);
 4: if (execution is granted) then
 5:    unprotect(shadow_symbol_table);
 6:    if shadow_symbol_table[fnct_name] not 0 then
 7:       address = shadow_symbol_table[fnct_name];
 8:    else
 9:       address = bind(fnct_name);
10:       shadow_symbol_table[fnct_name] = address
11:    end if
12:    protect(shadow_symbol_table)
13:    execute_function_at(address);
14:    apply_post_policies(fnct_name, params[]);
15: end if
```

One technical challenge that our reference monitor needs to tackle concerns the resolving of a function's name at runtime. This is required to accurately enforce policies at function-level. To address this challenge, we exploit the fact how functions are called in ARM-compiled binaries. In general, a function can be called by means of a `bl` (branch with link) or a `blx` (branch with link and exchange) instruction. Both instructions have in common that they store the return address (which is simply the address following the `bl` or `blx` instruction) into ARM's dedicated link register `lr`. Hence, when an application attempts to invoke an external function, it actually uses a `blx` instruction targeting the corresponding entry i in the code stub section:

```
1. blx code_stub_i
```

Since we instrumented the code stub section (i.e., by replacing `pc`-loads with branch instructions to our reference monitor), the reference monitor will be invoked next:

```
2. b reference_monitor
```

Next, we let our reference monitor read the value of `lr` as it contains the return address. This allows us to dynamically calculate the address from where our reference monitor has been invoked. As we now know from where inside the main application the reference monitor has been called, we can dynamically compute the target address of `blx code_stub_i`. Specifically, we decode the `blx` instruction on-the-fly to determine the targeted entry i in the code stub section. Once we know i, we are able to resolve the function name from a pre-computed table stored within the reference monitor:

```
3. decode_function_name(i)
```

This table is generated during static analysis of the binary and contains a list with the names of all external functions used by the iOS application. In addition, the equivalent dynamic loader code to resolve a function's name is integrated

[3] Previously denoted as `swi` (software interrupt) instruction.

into our reference monitor. We have implemented the reference monitor using ARM assembly instructions.

5.3 Shadow Table

Recall that within a normal executable the addresses of lazy-resolved functions are stored in the lazy symbol section. Our approach for protecting lazy-resolved functions' addresses is based on the following three steps: (1) the entries from the lazy symbol section are stored into the shadow table, (2) we zeroise the entries in the lazy symbol section, and (3) we protect the shadow table from being accessed outside the reference monitor. Note that we need to maintain a separate table due to limitations of existing memory management capabilities, i.e., memory operations such as changing access rights can only be done at page level. Hence, we cannot apply such actions directly to sections from the data segment of an iOS application without affecting adjacent data.

The entries in the non-lazy symbol section are populated by the dynamic loader before the program starts executing. In order to prevent attackers from reading these values at the beginning of program execution, we add a small startup code that we refer to as *pre-main* which setups the reference monitor and the shadow table, and zeroise the original lazy and non-lazy symbol section. To this end, we overwrite the `LC_MAIN` command in the iOS Mach-O header with the start address of *pre-main*. Hence, *pre-main* is the code which is executed first when an application is launched. For protecting the memory region of our shadow table, we invoke the well-known *mprotect()* system call, which allows us to mark the shadow table as readable when the reference monitor is executed, and as non-readable when program control is passed back to the application.

An adversary could try bypassing the *mprotect*-based shadow table protection mechanism through accessible memory management functions. However, *XiOS* checks the arguments (if they target the shadow table) and disallows these call to prevent leakage of the shadow table.

Note that our binary rewriting does not impact the main code section of the application. Thus our mechanism has no impact on the internal flow and performance of the application, and avoids error-prone binary rewriting.

5.4 Developer-Defined Policies

XiOS allows the creation and integration of developer-defined policies. While the first one is highly dependent on the application that the policies are made for, for the second one we have implemented an easy way to add policies. Specifically, the developer provides *XiOS* with two predefined functions, i.e., *pre_external_call* and *post_external_call*, that are executed before and after each external function call. These functions need to be implemented in C and self-contained, i.e., cannot use other external functions such as *memcpy* as this would end up intercepted by *XiOS* and create a loop. In order to facilitate the job of the developer we can provide her with a set of pre-defined functions, e.g., she can use *xios_memcpy* instead of *memcpy*.

Within the *pre_external_call* and *post_external_call* functions, the developer has access to the function name, arguments and return value. Note that the return of a function can be a value or a pointer referring to internal buffers. *XiOS* can handle both cases. However, each external function takes a different number of arguments and returns a different type; thus, the developer must be aware of the signature for the functions she aims to handle. Moreover, with *XiOS*, the return values and the arguments can be changed. The following code snippet shows an example where an external call to the *NSLog* function will not be allowed:

```
int pre_external_call(const char *function_name,
    const unsigned long regs[]) {
  char nslog[6] = ”NSLog”;
  if (xios_strcmp(function_name, nslog) == 0)
    return FAIL;
}
```

Note that the reference monitor does not differentiate between invocations of C functions or Objective-C methods, as the Objective-C mechanism dispatches all objects' methods through a generic C function called *objc_msgSend*. With *XiOS* both can be analyzed in the same manner within the two pre-defined functions.

Developing self-contained policies in C maybe cumbersome. Hence, we are currently working on a service where the developer submits the app, and uses a simple interface to define rules and triggers *XiOS*'s hardening mechanism. This will enable convenient definition of custom policies.

6. EVALUATION

In this section we present a detailed evaluation of *XiOS* with respect to its effectiveness and efficiency. First, we analyze the effectiveness of our hardening process against previous and our own new attacks (see Section 2.2 and 3). Then, we evaluate the performance impact of our changes on the application, and measure the overhead they impose on a randomly selected set of both C and Objective-C functions. Moreover, we evaluate our hardening mechanism on several real-world applications.

6.1 Effectiveness

In order to test the effectiveness of *XiOS* we use our malicious sample application that we introduced in Section 3. In particular, we modified our sample application so that it includes both attack types: (1) exploiting a runtime address of a public API ([19, 35] and Section 2.2), and (2) exploiting the runtime address of the dynamic loader function to resolve lazy symbols (Section 3). To this end, we first apply *XiOS* to our sample malicious application. Afterwards, we deploy the hardened application on our test iOS device (iPhone 4 running iOS 7.0).

Previous Attacks: To test *XiOS* against previous attacks, we let our malicious application de-reference the runtime address of a public API. Specifically, we invoke the `CGImageSourceCreateWithURL` public API residing in the public `ImageIO` framework as `ImageIO` contains references to *load-library* and *load-address*. Since the runtime address of the public API is lazily resolved on-demand, its runtime address will be stored in the lazy symbol section after we have called the public API during normal program execution. However, since *XiOS* completely overwrites the original lazy and non-lazy symbol section with zeroes, the de-referenced value will be always zero rather than the runtime address of `CGImageSourceCreateWithURL`. In *XiOS*, the runtime address of the public API is on the shadow table which is only accessible from the *XiOS* reference monitor. This effectively prevents the first attack step of previous attacks [19, 35].

Our New Attack: Recall that our improved attack does not require the knowledge of a public API runtime address.

	printf	mmap	strerror	NSString alloc	NSUserName	NSTemporaryDirectory
1 Policy Check	0.029459	-0.018641	0.016662	0.041377	-0.013764	0.096091
10 Policy Checks	0.037107	-0.002330	0.023010	0.063399	0.105262	0.058099
100 Policy Checks	0.140015	0.088113	0.091601	0.260605	0.334400	0.228264

Table 1: Overhead for 10,000 calls (in seconds)

Instead, our attack attempts to access the runtime address of the dynamic loader function `dyld_stub_binder` which is present in every iOS application in the non-lazy symbol section. *XiOS* successfully prevents this attack similar to the previous attack as it hides runtime addresses of external functions contained in the non-lazy symbol section by overwriting this section with zero and maintaining the runtime addresses in the protected shadow table.

To summarize, in both attack instances (previous and our new one), *XiOS* prevents the malicious application from determining critical runtime addresses of function pointers that could be exploited to invoke a private API.

Since the pointers in the lazy and non-lazy symbol section are no longer available to the adversary, he may try to leak function addresses by inspecting return buffers, return values and return register values in public APIs that can be called. We prevent this by the use of policies that inspect the return information of a public API and zeroise or replace any sensitive pointer data.

On the other hand, the adversary could perform a complete memory scan to identify private APIs and dynamic loader functions based on signature or instruction matching. However, this requires the adversary to know exactly the location of all mapped memory pages. Otherwise, the application will crash upon read access on non-mapped memory. Moreover, many consecutive read operations from main application code to shared library code resembles a program anomaly which one can certainly monitor at runtime.

A current limitation of *XiOS* are attacks through an address of a global external variable stored in the non-lazy symbol section. This allows an adversary to use offset-based computation to retrieve the address of dynamic loader functions. We successfully tested this attack and it is indeed currently feasible in iOS, since the shared library code and its data section are always located at the same offset, irrespective of ASLR. Such attacks are only possible as the ASLR scheme implemented in iOS does not randomize the offset between the data and code section. Since our focus in *XiOS* resides on attacks based on function pointers and more fine-grained ASLR solves the issue, we did not implement a mechanism to hide data pointers which is an orthogonal problem and we leave as future work.

6.2 Efficiency

Our hardening mechanism is deployed by the reference monitor for each function call. The overhead is constant irrespective of the type of function called and depends on the number of policies. As described in Section 4.3, we intercept only calls to external functions. Thus, the main application code and instructions from shared libraries execute with native performance.

In order to determine the overhead of our approach per external call, we have selected a variety of functions and evaluated their execution time. Specifically, we have implemented an application that calls only the tested function and implanted a set of 1, 10, 100 policies using *XiOS*. We measure the execution time when running the application with and without policy enforcement. Each of the selected functions runs for 10,000 times. Note that the selected policies only perform a sanity check (i.e., verify the function's name and print its arguments) and do not prohibit execution of functions as this would end the testing procedure.

We present the results in Table 1. The overhead for running Objective-C functions 10,000 times is, at most, only of 0.33s, while the overhead for native-C functions is only 0.14s in the case of 100 policies. The negative numbers displayed for *mmap* and *NSUserName* show that the hardened application was actually faster than the vanilla version. This is due to the fact that we do not control how the operating system scheduler (i.e., closed-source OS) affects our tests.

During our tests, we analyzed the average number of function calls per second inside an app. For normal use (clicks, accessing resources) there are no more than 1,000 function calls per second. This means, that if an app were to use only the functions listed in Table 1 the overhead would at most be 0.033s; less than 5%.

We applied our enforcements on a popular iOS benchmark tool called Gensystek [1]. Figure 5 shows results for eight different benchmarks with different number of threads (1, 2 and 4) and with different number of policy checks (1, 10 and 100). The computation of an MD5 hash displays the most noticeable slowdown of 3.9x. However, similar heavy-CPU computation such as Floating point calculation/Arithmetic logical unit (FPU/ALU) show an overhead of only 1.5% when tested against 100 policy checks. The overhead for taking and saving a screenshot to disk is 4.3%. Moreover, *XiOS* induces less overhead than previous work, e.g., MoCFI adds up to 500% overhead for the PI calculation [15]. Note that every external function called is applied the 100 policy checks, irrespective of what the function does, as these checks were part of the reference monitor implementation. Within a more realistic scenario, there would only be a handful of functions that do these many policy checks, reducing the overall incurred overhead.

In terms of required memory, our experiments with *XiOS* on real applications show that no additional space is required for storing the reference monitor inside the executable. This is due to the fact that the reference monitor requires only 1KB and can be stored in unused space of the binary, such as the `__TEXT` segment. The shadow table requires 4K and is allocated dynamically when the application is executed. In our evaluation, less than 1KB was required for one policy. However, if complex policies (hence, more space required for storing the policies) are to be inserted, new code sections and memory pages can be added at the end of the application.

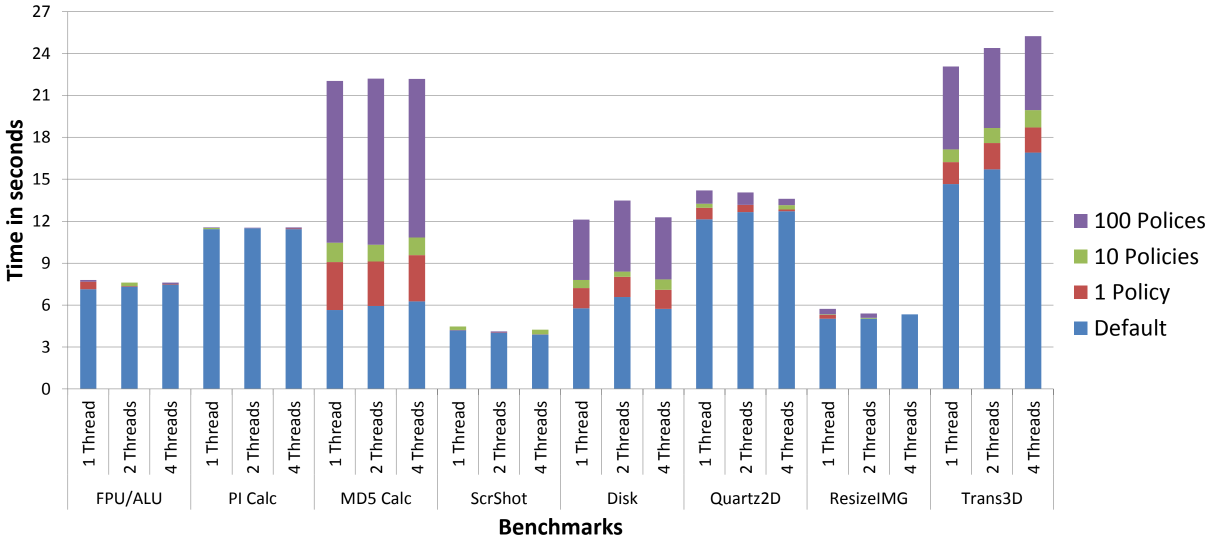

Figure 5: Gensystek Benchmark

No. of policies	SysMonitor	Frotz	MusicNotes
1	-8.78%	-9.05%	1.74%
10	3.68%	2.25%	-9.69%
100	21.39%	10.35%	-2.49%

Table 2: Loss in number of external functions calls

6.3 Study of Real World Apps

To demonstrate the effectiveness of our approach, we tested *XiOS* on several real-world applications such as Facebook, Twitter, Gmail, Youtube. In our experiences with the hardened applications we encountered no overhead. Moreover, we used our hardening service to apply policies to several applications such as System Monitor, Frotz, WhatsApp, MusicNotes and Gensystek. For testing the application behavior we sequentially used a number of 1, 10, and 100 policies and measured the number of external function calls with and without *XiOS* in a time frame of 30s. As before, policies are configured to allow all external calls. In order to access the application's binary (as iOS stores them encrypted), we made use of a well established tool, namely Clutch [14].

We compared ”vanilla” application runs against hardened application runs. The results are shown in Table 2 as the percentage of loss in the number of external calls. Remarkably, *XiOS* does not reduce the number of external calls that are made within a given time frame. A noticeable overhead of 21% is noticed only when *XiOS* is configured with 100 policies for each external function call which is far from realistic but represents worst-case scenarios. The MusicNotes application allows users to play different sounds at a touch of a screen-keyboard. We did not encounter any noticeable delay when performing different tests.

6.4 Access Control on Public Applications

XiOS allows enterprises to create and apply their own policies on any application available on the App Store. In order to demonstrate how such policies can be created we have chosen one of the most used application today, namely the instant messenger WhatsApp [3]. One particular feature of WhatsApp is that it uses the phone number for identifying different users in the network. Moreover, the full address book is copied to the developer servers when the application is installed. *XiOS* allows filtering of the contacts to avoid that WhatsApp retrieves all the user's contacts.

Before creating the filtering policies we need to identify which functions are relevant and need to be instrumented. To do so, *XiOS* provides a script that can be used to identify all external functions invoked by the application along with the framework that they belong to. Specifically, WhatsApp uses *ABAddressBookGetPersonWithRecordID* within the *AddressBook* framework to extract the contacts from the address book.

Next, the developer can check the function's signature and specify policies according to the format shown in Section 5.4. In our particular use-case we prevent WhatsApp from accessing all contact phone numbers that belong to the domain `corporate`. Note that the domain name is maintained in the *kABPersonOrganizationProperty* field belonging to the class *ABRecordRef*. Hence, when WhatsApp retrieves the contacts, we simply validate for each record whether *kABPersonOrganizationProperty* is set to `corporate`. If so, we simply replace the entire record with `NULL`. Appendix A presents the specific implementation of the policy.

7. RELATED WORK

The work related to *XiOS* can be roughly classified into protection schemes for iOS and runtime protection mechanisms that aim at hiding function pointers and instrumenting function calls.

iOS Security: There are a few proposals that aim at enhancing the security of iOS-based systems. PiOS is a static privacy analysis tool that performs off-line path validation on an application's control-flow graph [16]. Although PiOS revealed that many apps leak the device ID to application developers, it cannot detect Jekyll-like attacks [35] where the malicious behavior is only triggered at runtime. On the other hand, MoCFI [15] and PSiOS [37] could potentially prevent the mentioned attacks by enforcing control-flow integrity (CFI) and fine-grained sandboxing policies. However, both solutions require a *jailbreak* and suffer from performance problems, which limit their deployment in practice. In contrast, *XiOS* avoids jailbreaking devices and only incurs modest performance overhead.

Runtime Protection Mechanisms: For Android-based systems, several Inline Reference Monitors (IRM) have been proposed recently. Most of them insert a policy hook or check before a critical function is called at Dalvik Bytecode level [7, 22]. Most closely to our approach is Aurasium [39], as it deploys a similar redirection technique: it overwrites entries of the global offset table (GOT) – that holds runtime addresses of lazy and non-lazy symbols – with the start address of policy check functions. However, in contrast to *XiOS*, it does not provide any mechanism to hide the actual runtime addresses in a shadow table. Hence, an adversary can deploy memory disclosure attacks to infer the runtime address of a critical function and directly redirecting execution to it (via an indirect branch instruction). Moreover, Aurasium can be bypassed through native code that directly invokes system calls [13].

Since the GOT is critical in many Linux-based systems to intiate runtime attacks, Xu et al. proposed a solution that randomizes the GOT section at load-time, but allow its address to be discovered through the procedure linkage table (PLT) section [38]. Roglia et al. improve this solution by randomizing the GOT section and rewriting the PLT region [29]. The mechanisms behind those two solutions can be migrated to the iOS operating system. However, in contrast to our solution, both proposals need to be integrated into the operating system as they require higher privileges to rewrite a binary at load-time.

Another approach to instrument function calls aims at adding a wrapper and verifying the parameters passed to the function [9, 8]. A common mechanism to load such interceptors is the Linux `LD_PRELOAD` linker facility which forces the application to use a wrapper function rather than the actual external function. However, since it is unrealistic that every external function is replaced by a wrapper (due to space and complexity reasons), an adversary can exploit the knowledge of the runtime address of one single (not instrumented) function to directly call a private API.

In the domain of runtime attack mitigation the security model of address space layout randomization (ASLR) is used against memory (i.e., function pointer) disclosure attacks. The basic idea is to randomize the start address of code and data segments. Recently, several schemes have been proposed to even enforce fine-grained code randomization [10, 18, 21, 26, 23, 36, 12, 11]. However, sophisticated memory disclosure attacks [31] can bypass ASLR-based schemes. Moreover, in order to mitigate the attacks presented in this paper (Section 2.2 and 3), one would need operating system support and a jailbreak, because ALSR needs to be applied to the dynamic loader and public/private frameworks.

8. SUMMARY AND FUTURE WORK

Recent attacks have demonstrated that the current design of iOS is vulnerable to a variety of attacks that undermine the iOS sandboxing model leading to the invocation of private APIs (e.g., sending text messages in background). While previous attacks rely on specific assumptions such as the availability of a public framework, we showed that the default iOS application structure by itself can be easily exploited to invoke dangerous private APIs.

Since existing solutions suffer from performance overhead or require a jailbreak, we introduce a new hardening service, *XiOS*, that implants an inline reference monitor into an iOS application to tackle these attacks without requiring a jailbreak or source code of the application. This reference monitor efficiently prevents an application from inferring addresses of private APIs, and at the same time enforces (optional) developer-defined policies on public APIs. We demonstrate the benefits of the latter by enabling a contacts filtering mechanism for the popular WhatsApp messenger.

In the future, we plan to extend *XiOS* with a web frontend, where end-users can conveniently upload their custom policies that are automatically translated into C code and deployed at runtime when the application is launched on the device. In addition, we also aim at validating entire call chains rather than only enforcing access control on a per function-level.

9. REFERENCES

[1] Gensystek benchmark. `http://www.ooparts-universe.com/apps/app\_gensystek.html`.

[2] Lazy binding. `http://developer.blackberry.com/native/documentation/core/com.qnx.doc.neutrino.prog/topic/devel_lazy_binding.html`.

[3] Whatsapp. `http://www.whatsapp.com/`.

[4] Apple Inc. App review. `https://developer.apple.com/appstore/guidelines.html`.

[5] Apple Inc. iOS developer enterprise program. `https://developer.apple.com/programs/ios/enterprise/`.

[6] Apple Inc. iOS frameworks. `https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSFrameworks/iPhoneOSFrameworks.html`.

[7] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. AppGuard: Enforcing user requirements on Android apps. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.

[8] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference (ATC)*, 2000.

[9] A. Barenghi, G. Pelosi, and F. Pozzi. Drop-in control flow hijacking prevention through dynamic library interception. In *Tenth International Conference on Information Technology: New Generations (ITNG)*, 2013.

[10] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[11] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *USENIX Security Symposium*, 2003.

[12] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[13] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for

diverse security and privacy policies. In *USENIX Security Symposium*, 2013.

[14] Clutch. Clutch. https://github.com/KJCracks/Clutch.

[15] 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 *Network and Distributed System Security (NDSS)*, 2012.

[16] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Network and Distributed System Security (NDSS)*, 2011.

[17] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[18] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, 2012.

[19] J. Han, S. M. Kywe, Q. Yan, F. Bao, R. Deng, D. Gao, Y. Li, and J. Zhou. Launching generic attacks on iOS with approved third-party applications. In *Applied Cryptography and Network Security (ACNS)*, 2013.

[20] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS'98 Oper. Syst. Rev.*, 1998.

[21] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson. Ilr: Where'd my gadgets go? In *IEEE Security and Privacy (Oakland)*, 2012.

[22] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained permissions in Android applications. In *Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[23] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[24] H. Lu. ELF: From the programmer's perspective. http://linux4u.jinr.ru/usoft/WWW/www_debian.org/Documentation/elf/node7.html.

[25] Microsoft. Data Execution Prevention (DEP). http://support.microsoft.com/kb/875352/EN-US/, 2006.

[26] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Security and Privacy (Oakland)*, 2012.

[27] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.

[28] PGbiz. Count of active applications in the App Store. http://www.pocketgamer.biz/metrics/app-store/app-count/, 2014.

[29] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[30] 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.

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

[32] Symantec Corporation. 2013 internet security threat report, volume 18. http://www.symantec.com/security_response/publications/threatreport.jsp.

[33] T. Backdoor in top iPhone games stole user data, suit claims. http://www.theregister.co.uk/2009/11/06/iphone_games_storm8_lawsuit/, 2014.

[34] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[35] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on iOS: when benign apps become evil. In *USENIX Security Symposium*, 2013.

[36] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[37] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz. PSiOS: bring your own privacy & security to iOS devices. In *ACM SIGSAC Symposium on Information, Computer and Communications security (ASIACCS)*, 2013.

[38] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent runtime randomization for security. In *International Symposium on Reliable Distributed Systems*, 2003.

[39] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security Symposium*, 2012.

[40] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Security and Privacy (Oakland)*, 2009.

APPENDIX

A. WHATSAPP POLICY DEFINITION

```
unsigned long post_external_call(const char
    *function_name, const unsigned long
    returned_value)
{
  if (xios_strcmp(function_name, addressGetID) ==
      0) {
    char * companyName =
        ABRecordCopyValue(person,
        kABPersonOrganizationProperty);
    if (xios_strcmp(”corporate”, companyName) ==
        0){
      return 0;
    }
  }
  return returned_value;
}
```