iRiS: Vetting Private API Abuse in iOS Applications

Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, Dongyan Xu
Department of Computer Science and CERIAS
Purdue University, West Lafayette, IN 47907
{deng14, bsaltafo, xyzhang, dxu}@cs.purdue.edu

ABSTRACT

With the booming sale of iOS devices, the number of iOS applications has increased significantly in recent years. To protect the security of iOS users, Apple requires every iOS application to go through a vetting process called App Review to detect uses of private APIs that provide access to sensitive user information. However, recent attacks have shown the feasibility of using private APIs without being detected during App Review.

To counter such attacks, we propose a new iOS application vetting system, called iRiS, in this paper. iRiS first applies fast static analysis to resolve API calls. For those that cannot be statically resolved, iRiS uses a novel iterative dynamic analysis approach, which is slower but more powerful compared to static analysis. We have ported Valgrind to iOS and implemented a prototype of iRiS on top of it. We evaluated iRiS with 2019 applications from the official App Store. From these, iRiS identified 146 (7%) applications that use a total number of 150 different private APIs, including 25 security-critical APIs that access sensitive user information, such as device serial number. By analyzing iOS applications using iRiS, we also identified a suspicious advertisement service provider which collects user privacy information in its advertisement serving library. Our results show that, contrary to popular belief, a nontrivial number of iOS applications that violate Apple's terms of service exist in the App Store. iRiS is effective in detecting private API abuse missed by App Review.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813675 .

Keywords

iOS; Application Vetting; Private API; Forced Execution; Binary Instrumentation; Static Analysis; Dynamic Analysis

1. INTRODUCTION

Mobile devices, especially tablets and smartphones have gained tremendous popularity in recent years. Apple iOS is one of the dominating mobile platforms on the market; by the end of January 2015, Apple had sold one billion iOS devices [9]. One of its major success factors is the large number of third-party iOS applications that provide a wide variety of functionality to users. To rapidly grow the iOS ecosystem, Apple created the App Store which allows third-party developers to distribute their own iOS applications. As of September 2014, there were 1.3 million iOS applications available in the App Store [28].

Allowing third-party applications to run on iOS devices greatly improves the user experience. However, it also opens up the opportunity for malicious developers to attack the system and users. To prevent third-party applications from performing malicious activities, iOS employs several runtime protection mechanisms such as Sandboxing, Mandatory Access Control (MAC), Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR).

Unfortunately, even under these runtime protections, attack is still feasible through the use of private APIs. Private APIs are functions in iOS frameworks reserved only for internal uses in built-in applications. They provide access to various device resources (e.g. camera, bluetooth) and sensitive information (e.g. serial number, device ID), which are often not regulated by runtime mechanisms. Although some resources are guarded by entitlements with MAC in recent versions of iOS, there are still many that can be accessed without mediation.

As a countermeasure to the attack, Apple strictly prohibits any use of private APIs in third-party applications, according to its iOS developer license agreement [3]. To enforce the policy, every third-party application submitted to App Store must go through Apple's vetting process called App Review before it can be distributed to end users. Applications that pass App Review are digitally signed by Apple to prevent further modification. The signature is verified by iOS at runtime to ensure that only the original applications approved by App Review can run on iOS devices.

App Review has significantly raised the difficulty of distributing malicious applications to end users. Given the fact that very few malicious applications have been found on iOS [14], it is generally believed that App Review is quite

effective. However, recent work [15, 30] shows that by constructing the names of private APIs at runtime, it is possible to invoke private APIs in third-party applications and still be able to pass the vetting process. While Apple has never publicly disclosed the technical details of App Review, these attacks indicate that the current vetting process may be based on static analysis which is vulnerable to obfuscation. Although Apple complements automatic analysis with manual inspection [1], due to the large number of application submissions, it can only cover a small portion of all applications.

Besides Apple's App Review, there are several automated binary analysis systems [11, 29, 20, 18] proposed by security researchers to analyze iOS applications. However, these approaches also have shortcomings. Systems based on static analysis [11] could not resolve API names composed at runtime. Dynamic approaches [29, 20, 18] suffer from incomplete code coverage, thus would fail to detect uses of private APIs if malicious application authors place the invocations behind complicated triggering conditions.

To overcome the limitations of existing application vetting approaches on iOS, we present iRiS, an automated system that can effectively detect uses of private APIs in iOS applications. Given a binary iOS application, iRiS uses a combination of static and dynamic analysis to resolve the names of the functions being called in the program. iRiS first statically scans all function call sites and tries to resolve the names of the call targets using constant propagation and backward slicing. For the remaining call sites whose targets could not be statically determined, iRiS utilizes dynamic binary instrumentation to drive the execution of the application to the call sites to resolve the call targets at runtime.

We have encountered and solved many challenges of performing binary analysis on iOS in the design of iRiS. Due to the closed-source nature of iOS, there is no existing dynamic binary instrumentation framework available for it. As part of our effort, we have ported Valgrind [23] to iOS and built the dynamic analysis component of iRiS on top of it. Also, most iOS applications are based on event-driven graphical user interface (GUI) which exhibits very limited behavior without human interactions. In iRiS, we propose an automated UI event handler exploration approach by using dynamic binary instrumentation to monitor the registration of event handlers and trigger them automatically.

We have used iRiS to analyze 2019 free applications on the App Store. To our surprise, the results show that more than one hundred of these applications use private APIs. In some applications, we even identified the behavior of using private APIs to retrieve personal information (e.g. the applications installed on the device, the serial number of the device and its various components such as cameras and battery) and sending such information to advertisement providers. This suggests that the current application vetting approach used by Apple may not be sufficient to guarantee the security and privacy of iOS device users.

In summary, the contributions of our paper are as follows:

- We have ported the popular instrumentation framework Valgrind [23] to iOS. To the best of our knowledge, this is the first instruction-level dynamic binary instrumentation framework on iOS.
- We present the design and the prototype implementation of iRiS, an automated system using a combination of static and dynamic analysis to detect uses of private API in binary iOS applications.
- To show the effectiveness of our approach, we have analyzed more than 2000 iOS applications. Our result shows that a non-trivial number of iOS applications use security-critical private APIs to access and collect sensitive user information.

The rest of the paper is organized as follows. In Section 2 we introduce the background. We demonstrate the practical challenges and our solutions for porting Valgrind to iOS in Section 3. Then we present our approach of resolving API call targets in Section 4. We discuss the limitations of iRiS in Section 6 and compare with related work in Section 7. Section 8 concludes the paper.

2. BACKGROUND

In this section, we introduce background about various aspects of iOS. This will help readers to better understand our system described in later sections.

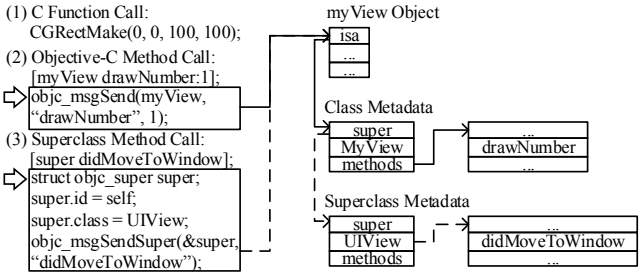

Figure 1: Different forms of function invocations in iOS applications.

2.1 Function Invocations

Objective-C is the major programming language used for building iOS applications. As an extension of the C programming language, Objective-C adds object-oriented features such as object, class and inheritance. In Objective-C, function invocations can take several different forms as shown in Figure 1. Since Objective-C is a superset of C, traditional C functions can be invoked as shown in case 1. In addition to that, Objective-C also supports object-oriented method calls as shown in cases 2 and 3.

The code in the boxes in cases 2 and 3 show how Objective-C method calls are actually implemented by sending *message* to *object* through one of the `objc_msgSend` family dynamic dispatch functions. More specifically, a message is composed of a *selector* which is the literal name of the method to be invoked, and the arguments to be passed to the method. In case(2), the `drawNumber` message is sent to the `myView` object. As shown in the path along the arrows, the `objc_msgSend` dispatch function locates the metadata of the object's class `MyView`, finds the implementation (i.e. entry address) of the `drawNumber` method and then calls it. Similar to other programming languages that support inheritance, if the corresponding method is not implemented in the object's class, the dynamic dispatch function searches through the object's superclasses along the class hierarchy.

Case 3 uses the `super` keyword to explicitly call a method in an object's superclass. An `objc_super` structure containing the `myView` object and name of its superclass `UIView` is

constructed and passed to the `objc_msgSendSuper` dispatch function. The dispatch function follows the dashed path to locate the `didMoveToWindow` method in `UIView` and call it.

The dynamic features of Objective-C grant iOS developers much flexibility in building their applications. Since selectors are just literal method names which contain no low-level information such as address, developers could easily construct selectors at runtime to send arbitrary messages to any object. Also, the mapping between selectors and method implementations could be modified at runtime. Such cases pose great challenges to binary analysis of iOS applications.

2.2 Private API

iOS provides a rich set of frameworks for building user-level applications. These frameworks are essentially directories that contain dynamic shared libraries and resources. The dynamic shared libraries expose APIs for applications in two forms: (1) as traditional C functions that are explicitly exported by the shared libraries; (2) as methods in Objective-C classes that are managed and dispatched by the Objective-C runtime.

Among all the frameworks, only some of them are public frameworks that are for use in third-party iOS applications. The other ones, known as private frameworks, are reserved for use in built-in applications and public frameworks only. Similar to frameworks, APIs are also categorized into public and private depending on whether they can be used in third-party applications. Note that public frameworks may also contain private APIs as part of their internal implementation.

Private frameworks and APIs provide many powerful functionalities that could threaten the security of the system if they are available to third-party applications. For example, the `SpringBoardServices` framework provides APIs to launch and terminate applications; the `IOKit` framework provides APIs to access mach I/O ports which could be used to obtain various device information. To prevent third-party developers from using private APIs, only public frameworks and APIs are documented and exposed by the header files in the iOS software development kit (SDK). However, despite efforts to conceal the prototypes of private APIs, they can still be reverse-engineered from the dynamic shared libraries in the frameworks [27].

Once their prototypes are known, calling private API functions follows the same procedure as calling public API functions. As a countermeasure, Apple requires every application submitted to the App Store to go through App Review to make sure the application binary is only linked to public frameworks and imports only public C APIs. Invocations of private Objective-C APIs are also detected because the `__objc_selrefs` section in the application binary contains all statically-known message selectors. However, such detection is not always effective. To evade the detection, an attacker may use the `dlopen` function to load private frameworks and the `dlsym` function to locate and call private C API functions. For private Objective-C APIs, the attacker can construct the message selectors at runtime so they do not appear in the application binary.

2.3 iOS Runtime Security

Similar to other modern operating systems, iOS incorporates standard runtime protections such as DEP and ASLR. In addition to that, it also implements several enhanced security mechanisms as described below.

Entitlements. iOS provides fine-grained access control that is based on the TrustedBSD MAC framework [32]. Each application can declare a set of *entitlements* that grant specific capabilities or security permissions in iOS. The iOS kernel checks for corresponding entitlements whenever an application is trying to access guarded resources. Most entitlements in iOS are for built-in applications; the only ones available to third-party applications are for enabling iCloud service and pushing notifications. To prevent third-party developers from abusing or counterfeiting entitlements, entitlements declared in third-party applications are checked for validity during App Review and then built in to the code signatures of the application binaries. Entitlements effectively regulate the use of private APIs: without proper entitlements; even if the attacker is able to invoke the private API, iOS will refuse the attempt to access the resource. Unfortunately, there are still many resources that are not protected by entitlements in iOS.

Prohibiting dynamic code generation. iOS disallows any kind of dynamic code generation, except for applications with the `dynamic-codesigning` entitlement. This entitlement is for the built-in `MobileSafari` application to implement its JIT Javascript engine and is unavailable to third-party applications. The prohibition of dynamic code generation in third-party applications has both a positive and negative impact on our system: it helps us to better disassemble third-party application binaries for static analysis because there is no dynamically generated or self-modifying code; on the other hand, it also disables dynamic binary instrumentation frameworks such as Valgrind due to their need to translate binary code at runtime. Fortunately, we can still port Valgrind to *jailbroken* iOS devices, where the kernel is patched to remove restrictions on dynamic code generation. Note, this does not indicate that the applications we analyze are also free to generate code at runtime; we still prohibit dynamic code generation in these applications by wrapping and checking the related system calls (e.g. `mprotect`) using Valgrind.

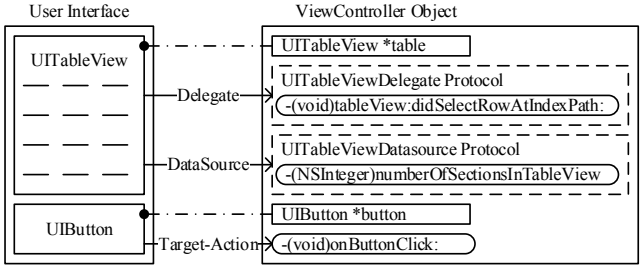

Figure 2: Event driven execution of iOS application.

2.4 Execution of iOS Application

We use an example in Figure 2 to demonstrate the execution of a typical iOS application. When the application is launched, it initializes a view controller object to create and manage views. In our example, the view controller object creates a `UITableView` object and a `UIButton` object to interact with the user. To handle user inputs, it sets delegate and data source for the table view and registers a

target-action event handler to the button. These are the two design patterns for implementing event handlers in iOS, which are described below.

Target-action. The target-action design pattern is used by all control classes (e.g. `UIButton`, `UITextField`) that are derived from the `UIControl` base class. Developers call the `addTarget:action:forControlEvents:` API to register a pair of *target* and *action* for a specific control event on a `UIControl` object. The action is the name of the Objective-C method to be invoked upon triggering the event and the target is the object that the method is called on. In our example, the application registers the `onButtonClick:` action with the view controller object as the target, for the click event on the button. When the button is clicked, the `onButtonClick:` method will be called on the view controller object.

Delegates and data sources. *Delegates* are objects that can be assigned to a view to provide application-specific event handling logic. When an event occurs, the view sends an Objective-C message to its delegate to invoke the corresponding event handler. Usually, a delegate must conform to the protocol corresponding to the view it is assigned to, so that the view knows the required methods are indeed implemented in the delegate.

In our example, the view controller object itself is assigned to the table view as a delegate to handle events such as selecting a row in the table. When a row in the table is selected, the `tableView:didSelectRowAtIndexPath:` method will be invoked on the view controller object. The view controller object conforms to the `UITableViewDelegate` protocol which declares the event handlers for table view.

Data sources are similar to delegates except they provide application-specific data instead of logic to views. In our example, the view controller object is also assigned to the table view as a data source. When iOS renders the table view, it invokes the `numberOfSectionsInTableView` method on the view controller object to determine how many sections are there in the table.

2.4.1 Nib Files

Besides creating views directly in the application code, iOS application developers may also choose to load UI elements stored in Nib (NeXT Interface Builder) files. Nib files are resource files generated by Apple's UI design tool called Interface Builder, which allows developers to design UI views and related non-visual objects (e.g. view controllers) in a visualized environment. The views and objects are serialized in the format of an object graph and stored in Nib files.

The UIKit framework provides several APIs to load Nib files at runtime. These Nib-loading APIs are responsible for reconstructing the views, objects and the connections among them to the same state as designed in Interface Builder. It is worth noting that in each Nib file, there is a special placeholder object called *File's Owner*. The File's Owner object is provided by the application as an argument to Nib-loading APIs, which serves as the link between the application code and the UI elements in the Nib file. It usually contains *outlets*, which are references to the views and objects in Nib files. The outlets are connected by the Nib-loading API during the process of loading Nib Files.

To demonstrate the details of the loading process, we still use the example in Figure 2, but we assume the views are loaded from a Nib file. We assume the information of the delegate, data source and the target-action event handler are all properly stored as connections to the File's Owner object in the Nib file. We provide the view controller object as the File's Owner object. According to Apple's documentation [5], the loading process consists of the following steps:

1. The Nib-loading API allocates the two view objects and sends them an `initWithCoder:` message to initialize the views. During the initialization of the table view, its delegate and data source are set to the File's Owner object, which is the view controller.

2. It connects the outlets (`table` and `button` variable) in the view controller to the two views by calling the `setValue:forKey:` method on the view controller. The values are the view objects and the keys are the name of the outlets.

3. It registers the `onButtonClick:` method in the view controller object as a target-action event handler to the button.

4. It sends an `awakeFromNib` message to the two views to notify them the loading is complete.

Clearly, the loading process implicitly involves invocations to many APIs, which all have to be considered in our analysis.

3. PORTING VALGRIND TO IOS

In order to build the dynamic analysis component in iRiS, we first ported the popular dynamic binary instrumentation framework Valgrind to iOS. Valgrind already supports ARM architecture. It also supports OS X, Apple's desktop operating system that shares the same kernel as iOS. Therefore, we could reuse the CPU-specific and OS-specific code. However, we still need to implement the parts that are specific to the combination of CPU and OS, which mainly include (1) the system call wrapper that executes system calls on behalf of the instrumented program; (2) the signal dispatcher that constructs the frames for signal handlers and (3) other OS-dependent code (e.g. Valgrind's bootstrap routine) that must be reimplemented in ARM assembly. In total, we added/modified over 6000 SLOC to Valgrind. We also encountered many practical challenges specific to iOS, some of which are discussed below. We plan to open source the ported framework to support future work on iOS security.

Calling convention of system calls. Valgrind needs to interpose system calls to perform many crucial operations, such as thread and memory management. The calling convention of system calls in iOS can be observed from the execution of the system call wrapper functions. More specifically, we build a program that calls system call wrapper functions with carefully crafted arguments. Then, we run the program and use GDB to set a breakpoint at those functions. Once a breakpoint is hit, we do single step until reaching a `SWI` instruction, which is used to perform system calls on ARM. It is then straightforward to infer the calling convention by observing which argument value is stored in which register or stack memory location at that point. We derived the calling conventions of all three types (BSD, MACH and MDEP) of system calls in the iOS kernel using this approach.

Reading symbols from dyld shared cache. The symbol table maintained by Valgrind is important for translating addresses to human-readable API names. Normally, Valgrind reads symbols from shared libraries when they are loaded into the address space of the application. However, there is no such loading of individual libraries in iOS. All shared libraries in iOS are combined into a single large file called dyld shared cache, which is mapped into the application's address space by the kernel when the application is loaded. To read the symbols, we invoke the `shared_region_check_np` system call to obtain the start address of the shared cache. Since the symbols of all libraries are too large to fit in the memory available to Valgrind, we read the symbols of a specific library from the shared cache only when its code is executed the first time.

Instrumenting GUI applications. In iOS, GUI applications have to be launched by sending a launch request with the bundle id of the application to `SpringBoard`. Clearly, Valgrind has to be launched this way when instrumenting GUI applications. However, the applications launched by `SpringBoard` run on behalf of the user `mobile`, which does not have the root privilege required by Valgrind. We solve this problem by setting the owner of the Valgrind executable to `root` and setting its `setuid` attribute.

4. RESOLVING API CALL TARGETS

4.1 Overview

The goal of iRiS is to identify the targets of all API calls in iOS application binaries. This cannot be done with pure static analysis due to the dynamic features of Objective-C. Theoretically, dynamic analysis could resolve all the targets by utilizing approaches such as symbolic execution [19] or forced execution [25] to explore every path leading to an API call. However, such approaches are infeasible in practice due to the large size of iOS applications. For example, the Facebook iOS application binary is sized at 48MB, containing about 10 million instructions and 1.4 million branches. Binaries of such scale could not be handled by symbolic execution. Even forced execution with complexity linear to the number of branches would need several weeks to explore all the necessary paths in a single application.

To solve this problem, we adopt an approach that combines static and dynamic analysis in iRiS. Our key observation here is that the vast majority of call targets in normal iOS application binaries can be resolved using static analysis, which is fast and scales well with the size of the program. For the very few remaining call sites whose targets cannot be statically determined, we apply the slower, but more powerful dynamic analysis to get the targets from the concrete execution states at the call sites.

An overview of iRiS is shown in Figure 3. The input to iRiS is an iOS packaged application (with an .ipa file extension) downloaded from the App Store, which is essentially a zip file containing the application executable, resources and other metadata. iRiS first extracts the application executable and the Nib resource files from the package. Since all applications submitted by third-party developers are encrypted by Apple before they are distributed through the App Store, iRiS needs to decrypt the application executable to the raw binary executable before it can proceed to the analysis.

The analysis begins with resource analysis of the Nib files. For each Nib file, iRiS identifies the functions in the application binary that are implicitly invoked when the Nib file is loaded. In this way, each Nib file is represented as a set of call targets it implies. In later stages of static and dynamic analysis, upon encountering an API call that loads a Nib file, iRiS will add the call targets implied by the Nib file to the API call site.

After analyzing Nib resources, iRiS performs static analysis on the decrypted application binary executable. iRiS disassembles the binary using IDA Pro [16] and scans for all call sites. Similar to PiOS [11], iRiS tries to use backward slicing and forward constant propagation to resolve the call targets at each call site to generate an initial call graph. For each function in the binary, iRiS also generates its intra-procedural control-flow graph (CFG). The initial call graph and intra-procedural CFGs serve as guidance for the final stage of analysis.

In the final stage, iRiS iteratively resolves the remaining call sites whose targets could not be statically determined, using dynamic analysis. In each iteration, iRiS picks a call site with unresolved targets from the call graph, and uses the call graph and intra-procedural CFGs to explore paths to the call site to obtain the call targets. The resolved call targets are merged back to the call graph, which helps with resolving more targets in later iterations. After all iterations are finished, the call targets in the final call graph are checked against iOS SDK headers to reveal uses of private APIs.

4.2 Resource Analysis

Resource analysis aims to identify application functions implicitly invoked in the process of loading a Nib file. It is infeasible to statically examine a Nib file to obtain such information since the file format is not publicly known. Our idea here is to load the Nib file artificially using the API in the `UIKit` framework, and use Valgrind to monitor the function invocations in this process.

However, there are several challenges to load a Nib file artificially. Creating a dummy program that blindly calls the Nib-loading API would most likely fail, as a Nib file is not a self-contained entity that can be loaded in an arbitrary context. For example, the objects stored in a Nib file might be of custom classes defined in the application binary. The Nib-loading API would fail when it tries to invoke the initialization methods of these objects, as they do not exist in the dummy program. Also, since the provided File's Owner object does not contain the outlets expected in the Nib file, the Nib-loading API would fail when trying to connect the outlets.

To overcome these challenges, we utilize the application itself to provide the proper context for loading its own Nib files. We run the application with `DYLD_INSERT_LIBRARIES` environment variable to inject a preload shared library containing the Nib-loading code to its address space. In the preload shared library, we put the invocation of the Nib-loading API in a function with the `constructor` attribute so it is executed before any other code (except global initialization routines) in the application binary. To handle outlets, we provide a fake File's Owner object to the Nib-loading API which ignores connections to undefined outlets by overriding the `setValue:forUndefinedKey:` method, which is the fail-safe method when the `setValue:forKey:` method for

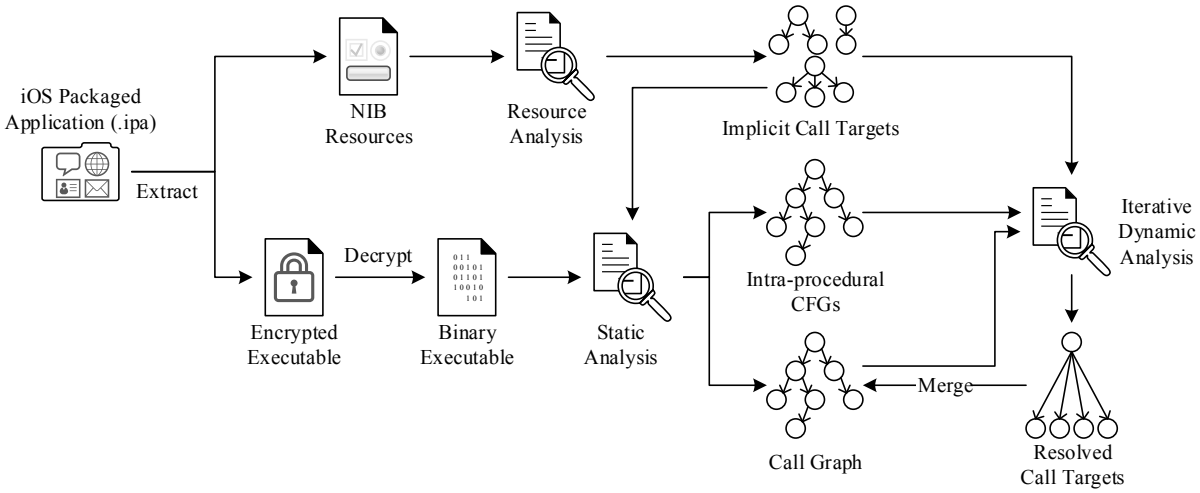

Figure 3: Overview of iRiS.

connecting outlets fails. We terminate the application by calling `exit` right after the Nib file is loaded so no unrelated code is executed.

Event handler registration functions need to be handled specially. Although the event handlers are not directly called when they are registered during Nib loading, we include them as implicit call targets so that they can be explored later in the iterative dynamic analysis stage. Since we have the concrete execution state, we can query the Objective-C runtime to get the entry addresses of the event handlers (as the parameters to the registration functions). A target-action event handler is identified if the method has the *action* selector implemented in the class of a *target* object. For delegate or data source, we enumerate the methods that are implemented in the class of the delegate or data source object and include the ones listed in the delegate or data source protocol.

Another case that requires special handling is the function invocation to connect outlets. The `setValue:forKey:` method for connecting outlets internally invokes the setter methods of the File's Owner object to set its properties. However, since we artificially load the Nib file by providing a fake File's Owner object, the expected type of the real File's Owner object is unknown and the entry addresses of the setter methods could not be determined at this time. Therefore, we record the keys that are being set here so that the setter methods can be resolved when the class of the File's Owner object is known at later stages of analysis.

The final step of resource analysis is to prune the implicit call targets that have been obtained so far. This is because the Nib-loading API calls other functions in the `UIKit` framework or other frameworks. Such invocations might target private APIs, which is normal for internal interactions between iOS frameworks but would trigger false alarms if included in our result. We exclude the call targets that are not functions in the application by checking whether they fall in the range of the code section in the application binary.

4.3 Static Analysis

The goal in the static analysis stage is to build the intra-procedural CFGs and resolve call targets to construct call graphs. We build our static analysis as a plugin of the popular IDA Pro disassembler. Generating the intra-procedural CFGs is straightforward as IDA Pro already performs intra-procedural flow analysis for each function. However, the ability of IDA Pro to resolve call targets is quite limited. For traditional C function calls, IDA Pro can only identify direct call targets represented as constant relative addresses embedded in the instructions; it does not resolve indirect call targets that are stored in registers. Moreover, IDA Pro does not resolve function arguments stored in either register or stack variables. They are especially important for analyzing the target of Objective-C method invocations. For example, even if a call to the `objc_msgSend` message dispatching function is identified, we would not be able to know the exact Objective-C method being invoked unless we resolve the message selector and the object class type from the arguments of the function.

To resolve the call targets that cannot be handled by IDA Pro, we build our analysis based on the approach proposed in PiOS [11] which consists of intra-procedural backward slicing and forward constant propagation. The basic idea is to use backward slicing to recursively identify a slice of instructions that influence the value of the register or stack variable related to the call target at the call site. Starting from the beginning of the slice, statically known constant values are propagated forwardly according to the semantic of the instructions in the slice to compute the target value.

Our static analysis consists of three passes on the application binary. Compared with the original approach in PiOS, our approach covers more forms of Objective-C message dispatching and handles implicit invocations which result in a more precise and complete call graph. The details of each pass are described below.

4.3.1 Resolving C Function Calls

In the first pass, we identify all traditional C function calls and resolve their call targets. On ARM architecture, functions calls are made with `BL` (branch with link) and `BLX` (branch with link and exchange) instructions. We enumerate all these instructions in the application binary and check their operands. Constant operands representing direct call targets are already identified by IDA Pro. For register operands that contain indirect call targets, we try to use backward slicing and forward constant propagation to resolve their values. For those unresolved operands, we mark

the corresponding call targets as unknown. A resolved call target is identified as an external API if the target address is one of the following two cases: (1) the address of an API in the imported symbols section or (2) the address of a *stub* function that is a trampoline for calling an external API.

4.3.2 Resolving Objective-C Messages

Calls to Objective-C message dispatching functions (e.g. `objc_msgSend`) are identified in the first pass. In the second pass, we try to resolve the actual Objective-C methods invoked in those message dispatching function calls.

For the message dispatching functions that invoke methods in the object's class, such as `objc_msgSend`, we use backward slicing and forward constant propagation to resolve the message selector and the object's class in the function arguments. Similar to PiOS, to resolve the object's class, we propagate not only constants, but also type information along the slice. Once the message selector and the object's class are resolved, we find the corresponding method in the class hierarchy obtained from the application using the `class-dump` [24] tool.

Other dispatching functions, such as `objc_msgSendSuper`, are used to explicitly invoke methods in object's *superclass* (Section 2.1). The name of the superclass is provided in an `objc_super` structure, which is pointed to by one of the function arguments. To identify the superclass, we apply two rounds of slicing and constant propagation: the first one resolves the argument pointing to the `objc_super` structure and the second resolves the superclass name in the structure. In most cases, the values could be successfully resolved as these functions are mainly inserted by the compiler to handle the `super` keyword in Objective-C source code, where the superclass is known at compile time.

Any Objective-C method that is not successfully resolved here is marked as unknown target to be processed later in iterative dynamic analysis.

4.3.3 Resolving Implicit Invocations

We resolved the targets of explicit C function calls and Objective-C method invocations in the previous two passes. In the final pass, we aim to find and resolve the targets of implicit function invocations, which are categorized and discussed as below.

Grand central dispatch. Grand central dispatch (GCD) is a runtime system to support concurrent code execution on iOS. It provides APIs (e.g. `dispatch_async`) for developers to submit functions to dispatch queues for execution. The argument of a GCD API could be a function pointer or a block object (a wrapper structure for a function pointer). In either case, we apply backward slicing and forward constant propagation to get the address of the submitted function and add it as a call target.

Objective-C runtime. As we have mentioned in Section 2, the implementation of an Objective-C method can be changed at runtime. Therefore, it is possible for a malicious application developer to define a placeholder method, and replace its implementation with a private API function. After that, the developer could invoke the completely legitimate placeholder method to use the functionality of the private API. To prevent such attacks, we try to resolve the arguments of all functions in the Objective-C runtime that are related to retrieving or replacing the implementation of a method (e.g. `class_replaceMethod`). Although retrieving the implementation of a private API does not necessarily mean it will be called, we still consider any such behavior to be a violation due to the complexity of reasoning about method replacement statically.

Nested message passing. Some Objective-C classes provide methods to send messages, which resembles the functionality of `objc_msgSend`. For example, `NSObject`, the root class of all other Objective-C classes, provides the `performSelector` family methods which allow an object to send a message indicated by the argument to itself. The message could even be another `performSelector` which results in nested message passing. To handle such cases, we resolve the function arguments recursively until we reach the innermost message, which is added as the actual target.

Event handler registration. Similar to resource analysis, when we identify an event handler (target-action, delegate or data source) registration, we add the event handler as a call target so it could be explored later in dynamic analysis.

Nib file loading. When a call to a Nib-loading API is identified, we try to resolve the name of the loaded Nib file in the function argument. Once we know which Nib file is loaded, we add its corresponding implicit call targets obtained in resource analysis to the call site of the Nib-loading API. We also resolve the class of the File's Owner object provided to the Nib-loading API. In resource analysis, we could not resolve the setter methods of the File's Owner object that are invoked to connect outlets, because the class of the File's Owner object is unknown at that time. With the concrete class of the File's Owner object here, those methods can be resolved and added as implicit call targets.

The Nib-loading APIs in the `UINib` class are handled specially as they consist of two steps to load a Nib file. First the `nibWithNibName:bundle:` method is called to cache the Nib file in memory, and the Nib file is loaded at a later time using the `instantiateWithOwner:options:` method. Since it's infeasible to statically correlate the calls to these two methods, we leave them to be handled in dynamic analysis.

4.4 Iterative Dynamic Analysis

In the final stage of the analysis, iRiS uses dynamic analysis to resolve the call targets that cannot be determined in the static analysis stage. In dynamic analysis, as long as a function call is covered in an execution, it is straightforward to get its target and arguments from the concrete execution state at the call site. However, the task of reaching a specific call site in a dynamic execution itself is challenging. Also, we have to solve the problem of exploring the program paths that can affect the target and arguments of the function call.

We propose an iterative algorithm to find and explore the paths that could reach the target function call sites, as shown in Algorithm 1. The exploration is based on the initial call graph and the control-flow graphs of all functions generated by the static analysis. Initially (line 1), the application binary is directly executed in Valgrind without user interaction to record all call sites in the call graph that are covered in the natural run. These call sites serve as our starting points in the following rounds of exploration. The algorithm then explores the paths and updates the call graph in each iteration (line 4 to line 12). It terminates when there is no change to the call graph after an iteration (line 13).

In each iteration, we process each unresolved call site individually (line 6 to line 11). We denote that there is a *transition* from a call site cs_A to another call site cs_B if the

Algorithm 1 Call Targets Resolving Algorithm

Input:	CS - the set of unresolved call sites in static analysis CG - the call graph produced by static analysis CFG - the intra-procedural control-flow graphs produced by static analysis
Output:	CG - the updated call graph with edges to newly resolved call targets

1: $CS_n \leftarrow$ {call sites covered in the natural run}
2: $CS_{prev} \leftarrow$ {{nil} * sizeof(CS)}
3: **repeat**
4: $change \leftarrow$ {nil}
5: **for** $i \leftarrow 0$ **to** sizeof(CS) **do**
6: $CS_{rel}[i] \leftarrow \{cs_r \in CG \mid \exists cs_n \in CS_n : cs_n \rightsquigarrow cs_r \rightsquigarrow CS[i]\}$
7: **if** $CS_{rel}[i] \setminus CS_{prev}[i] \neq \emptyset$ **then**
8: $targets \leftarrow$ ForceExecute($CS_{rel}[i], CS[i]$)
9: $change \leftarrow change \cup$ InsertTargets($CS[i], targets, CG$)
10: $CS_{prev}[i] \leftarrow CS_{rel}[i]$
11: **end if**
12: **end for**
13: **until** $change = \emptyset$

function f_B that contains cs_B is one of the call targets at cs_A. With these transitions as edges, the call sites forms the graph CG. Given an unresolved call site cs_A, we first compute its *related* call sites, which are the call sites along the paths from any call site in CS_N to cs_A(line 6). These related call sites are the ones that we use to guide the natural execution to the target unresolved call site. If the set of related call sites is different from the one in the previous iteration (line 7), the algorithm will explore paths following the new guidance to identify potential new targets at the call site (line 8).

The `ForceExecute` function (line 8) to explore paths is based on the path exploration algorithm in X-Force [25]. X-Force forces control-flow at branches to explore the basic blocks in a program. In our scenario, the call sites are analogous to the basic blocks. The transitions from one call site to another are analogous to the branches at the end of the basic blocks. We force those transitions to explore paths along related call sites. The application runs naturally at the start of each execution of the exploration. Once the execution reaches any related call site, we start forcing transitions. Unlike in X-Force, where the exploration is unbounded, we limit the transitions to *related* call sites in our exploration, which ensures that each execution eventually reaches the desired unresolved call site to get its call targets. For the purpose of demonstration, let us assume the execution currently reaches the call site cs_A, which calls the function f_B. To force a transition from cs_A to a call site cs_B in f_B, we force the control-flow from the entry of f_B to cs_B by forcing branch targets in f_B. We compute the basic blocks in the paths from the entry basic block of f_B to the basic block containing cs_B in the control-flow graph of f_B, which we denote as *safe* basic blocks since execution reaching any other basic block will not be able to reach cs_B. In the execution starting from the entry of f_B, at each branch, we force the branch target if it does not fall in the set of the *safe* basic blocks. In this way, we guarantee the execution will reach the call site cs_B with as few forced branches as possible.

There are some cases that need to be handled specifically during the exploration, which are discussed below:

Event handlers. In static analysis, event handlers are added as the call targets of their registration call sites. However, this is only for the purpose of path exploration algorithm; the event handler itself is not actually invoked at its registration site. Directly manipulating the call target at the registration call site to force a call to the event handler will most likely fail because it does not provide the proper context for the execution of the event handler. Therefore, the exploration of each event handler has to be handled based on its type:

- **Target-action**. A target-action event handler is registered as a pair of *action* selector and *target* object on a `UIControl` object. To trigger the event handler, we use `dispatch_async` to dispatch a call to the `sendAction:to:forEvent:` method on the main dispatching queue of the program. When the call is dispatched, the `UIControl` object sends a message with the *action* selector to the *target* object.
- **Delegates and data sources**. First, we construct an `NSInvocation` to artificially invoke a specific event handler implemented by a delegate or data source. The target of the `NSInvocation` is set to the delegate or data source object, and the selector is set to the name of the event handler. The first argument is the `UIView` object which the delegate or the data source is assigned to. We pass zero to all other arguments by allocating a zeroed buffer on the stack that is as long as the size of the remaining arguments. The `NSInvocation` we construct is then dispatched on the main dispatching queue of the program.

Nib file loading with UINib. As we mentioned in Section 4.3.3, the `UINib` class involves two steps to load a Nib file. In the first step, we track the call to the `nibWithNibName:bundle:` method to record the name of the Nib file cached in the `UINib` object. When the program later calls `instantiateWithOwner:options:` on a `UINib` object to load the cached Nib file, we refer to the recorded information to get the name of the corresponding Nib file. At this time, we can resolve the calls involved in the Nib-loading process as both the Nib file name and the owner object are known.

Once the exploration has finished, the revealed call targets at the unresolved call site will be merged into the current call graph (line 9). Theoretically, the complexity of exploration of all possible paths is exponential to the number of related call sites. In practice, we support a number of exploration strategies (e.g. linear and quadratic) with different trade-offs between completeness and complexity. In our current implementation, we choose to use the linear complexity exploration strategy. When performing a round of forced executions to resolve an unresolved call site (line 8), we only force transition to a related call site if it has not been covered in previous executions in this round. Therefore, the number of executions is linear to the number of related call sites.

5. EVALUATION

We evaluated iRiS on 2019 free applications obtained from one of the largest official iOS App Stores. These applications are the ones listed as *popular apps* in the following categories in iTunes preview [4]: education, entertainment, finance, fitness, lifestyle, medical, productivity, social and utility. We

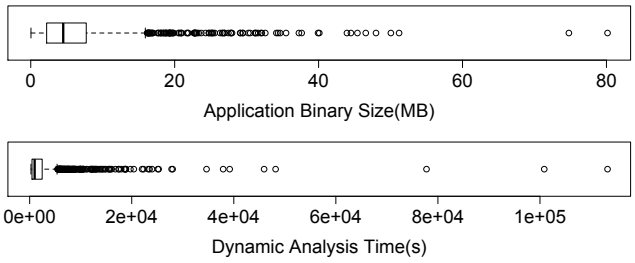

Figure 4: Distribution of the application binary size and the dynamic analysis duration.

crawled the iTunes preview website to retrieve the item ids of these applications. We downloaded the applications through iTunes and decrypted them using the `dumpdecrypted` [13] tool. We analyzed the applications using an iPad 3 16GB and an iPad 4 16GB, both running iOS version 7.0.

During the process of using iRiS to analyze the 2019 applications, we have identified 135,682,132 Objective-C related calls (including 135,650,013 `objc_msgSend` family message dispatching functions and 32,119 other Objective-C runtime functions), and 11,266 invocations to other C functions such as GCD APIs. Our static analysis was able to resolve the C function names or the Objective-C message selectors for 135,653,346 call sites (99%). For Objective-C related call sites, iRiS was able to identify the class of the corresponding object for 115,763,581 call sites (85%), which is similar to PiOS (82%). Note that even when the class of the object could not be resolved at a call site, it would not target a private API if the message selector did not match the name of any private API. In such cases, we did not need to apply further dynamic analysis on that call site. In fact, we did not find any private API name in the statically resolved C function names or Objective-C message selectors.

For the remaining 40,052 call sites that were not resolved in static analysis, iRiS was able to resolve the target for 35,427 of them (88%) with iterative dynamic analysis. Note that although the total number of call sites to resolve in dynamic analysis is small (less than 1% of all identified call sites), they exist in 1859 (92%) of the 2019 applications, which confirms that dynamic analysis is required in vetting most of the iOS applications. The natural runs of the applications covered 18671 call sites (47%), and the remaining 16756 resolved call sites (41%) are explored with forced executions. The duration of the dynamic analysis varies depending on the number of call sites to resolve and the complexity of the application, with the shortest one taking 257 seconds and the longest one taking 113,241 seconds to finish. On average, it takes 2439 seconds to analyze one application in our evaluation set. The distribution of the application binary size and the dynamic analysis duration is presented in Figure 4. Note that for universal binaries that support multiple architectures, we use Apple's `lipo` tool to extract the binary for the ARMv7 architecture.

Among the 2019 applications, iRiS identified 146 applications that contain invocations to a total of 150 different private APIs. We manually inspected these applications to understand the strategies used by their authors to circumvent the App Review. The most commonly used approach is concatenating or formatting API names and message selectors using `NSString` class methods (e.g. `[NSString stringWithFormat:]`) and traditional C string manipulation functions (e.g. `strcat`, `snprintf`). We even found one application defining a dedicated function to make private API calls; the function concatenates two parameters together to make a message selector and then invokes the method, probably to thwart static intra-procedural data-flow analysis. Some applications use more advanced obfuscation, such as decryption (e.g. XOR, RC4) to decrypt selectors that are encrypted in the application binary.

Many of the private APIs we identified are for implementing non-standard user interface features. For example, several applications use the `setOrientation:` method in the `UIDevice` class to force the orientation of the device display. Although uses of such APIs also violate the iOS developer license agreement, we will not discuss them in detail here since they are not directly related to security. The remaining invoked private APIs, which warrant further scrutiny for suspicious behavior, are categorized and shown in Table 1 and discussed below.

Accessing Application Information. `SpringBoardServices` is the framework that handles application launching, management and termination on iOS. It contains various APIs to query the status of the applications on the device. We found three applications using these APIs to obtain the bundle identifiers of the currently running and the front most application(s). After the bundle identifiers are retrieved, they are translated to application names by calling another private API. We also observed another 30 applications that call the bundle id translation API. The translation API returns a NULL pointer for non-existing bundle id, which is used by those applications to detect whether a specific application exists on the device.

We also identified two applications using private APIs in the `LSApplicationWorkspace` class of the `MobileCoreServices` framework to obtain the information of all applications installed on the device. The use of the `allApplications` API to get the bundle id list of all installed applications is also mentioned in a recent work [35]. We speculate that the two applications use these APIs instead of the private APIs in the `SpringBoardServices` framework because the latter ones are blocked by Apple since iOS 8.

Accessing User Identification Information. We found one application that invokes the `appleIDClientIdentifier` API in the `AADeviceInfo` class to obtain the Apple ID of the current user. Also, there are 25 applications using the APIs in the `ASIdentifierManager` class to obtain the Advertising Identifier (AdID) of the device. AdID is an identifier which could be used to uniquely identify an iOS device. It serves as the replacement of the unique device identifier (UDID) for advertisement serving organizations after access to the UDID was disabled in iOS 7. As mentioned in Apple's documentation [2], AdID should only be accessed by advertisement serving libraries (e.g. Google AdMobs). However, we found that the `crashlytics` library, which is a library for crash reporting, calls these private APIs to access AdID in these 25 applications.

We also found 21 applications using private APIs exported by the `IOKit` framework to access various hardware information. The `IOKit` framework is for communication with low-level hardware on the iOS device. It exports various hardware components as a tree of `IOService` objects. We found that 19 of these applications use the `IORegistryEntryCre-`

Category	Framework	API Name	Functionality	#apps
Access Application Information	SpringBoardServices	SBSSpringBoardServerPort	Initialize port with SpringBoard	3
		SBSCopyApplicationDisplayIdentifiers	Obtain bundle ids of all running apps	3
		SBFrontmostApplicationDisplayIdentifier	Obtain bundle id of the front most app	3
		SBSCopyLocalizedApplicationNameForDisplayIdentifier	Get app name from its bundle id	33
	MobileCoreServices	[LSApplicationWorkspace defaultWorkspace]	Obtain the default workspace object	2
		[LSApplicationWorkspace allApplications]	Get all installed apps	1
		[LSApplicationWorkspace allInstalledApplications]	Get all installed apps	1
		[LSApplicationWorkspace applicationIsInstalled:]	Check if a specific app is installed	1
Access User Identification Information	AppleAccount	[AADeviceInfo appleIDClientIdentifier]	Obtain the Apple ID of the device user	1
	AdSupport	[ASIdentifierManager sharedManager]	Obtain reference to the AdID manager	25
		[ASIdentifierManager advertisingIdentifier]	Obtain the device's AdID	25
		[ASIdentifierManager advertisingTrackingEnabled]	Check if advertising tracking is enabled	23
	IOKit	IOMasterPort	Initialize communication with IOKit	21
		IOServiceMatching	Find & open specified IOService object	21
		IOServiceGetMatchingService		21
		IORegistryEntryCreateCFProperty	Locate specific property (e.g. S/N)	19
		IORegistryEntryCreateCFProperties	Iterate through all properties to find information (e.g. Battery id, IMEI)	2
		IORegistryGetRootEntry		2
		IORegistryEntryGetChildIterator		2
		IOIteratorNext		2
		IORegistryEntryGetNameInPlane		2
		IOObjectRelease	Release the IOService object	2
Access User's Data/Settings	UIKit	[UIStatusBarServer getStatusBarData]	Get precise battery level	1
		[UIView createSnapshotWithRect:]	Capture the view as an image	1
Anti-debugging	libsystem	ptrace	Prevent GDB attaching	1

Table 1: Uses of private APIs detected by iRiS in iOS applications.

`ateCFProperty` API to read the serial number of the device from the `IOPlatformSerialNumber` property in the tree of `IOService` objects. The other two applications use a set of private APIs in `IOKit` to iterate through the tree of `IOService` objects to find the desired information. We manually inspected these two applications and found out that they try to obtain the ID of the battery and the serial numbers of the front and back camera by looking for the properties of specific names. Further investigation reveals that the serial number of the iOS device itself is protected by entitlement since iOS 8; however, the identification information of the battery and cameras are still available.

Accessing User's Data/Settings. We identified two applications using private APIs in the `UIKit` framework to access sensitive user data. One of them tries to obtain the current battery level from the status bar. According to our investigation, this private API allows the application to get the precise battery level compared with using the `batteryLevel` public API in the `UIDevice` framework, which only rounds the battery level to the nearest 5%. The other application calls another private API in the `UIView` class which allows the application to capture the displayed content in a view and save it as an image.

Anti-debugging. We found that one popular application calls the `ptrace` function with the `PT_DENY_ATTACH` argument to prevent itself from being attached by GDB. Since `ptrace` is a private API that is not declared in the header files in iOS SDK, the application calls `dlsym` to dynamically retrieve the entry address and then make a call to the function.

5.1 Case Study: A Suspicious Advertisement Service Provider

In this case study, we discuss our experience of identifying a suspicious advertisement service provider that collects user privacy information from various iOS applications in the App Store. Our finding started from the analysis of a utility application, anonymized as APP_S. The size of the application binary is 3.21 MB and its disassembly produced by IDA Pro contains 709,894 instructions.

We used iRiS to perform a thorough analysis of this application. In the static analysis stage, iRiS identified a total number of 210,534 call sites (excluding the ones in API call stubs), in which 52,814 were Objective-C message dispatching calls. iRiS successfully resolved most of the call targets in the static analysis; there were 21 unresolved call sites left to be examined in the iterative dynamic analysis stage. Despite the large number of statically resolved call targets, none of them actually pointed to any private API.

iRiS first performed a natural run of the application in the dynamic analysis stage. In the natural run, 13 of the 21 unresolved call sites were covered; 8 of them were targeting private APIs. The private APIs being called were the ones in the `SpringBoardServices` framework for accessing application information and the ones in the `IOKit` framework for accessing the serial number of the device shown in Table 1. There were also three of them calling the APIs in the `AdSupport` framework to get the AdID of the device. However, because our later analysis shows those three calls are in an advertisement serving library, we do not considered them as private API calls.

The remaining 8 call sites not covered in the natural run were resolved iteratively with forced execution. Two of them target private APIs in the `MobileCoreService` framework for obtaining the bundle ids of all installed apps; the rest of the call targets are functions in the application binary. We closely examined the two private API call sites and found that they shared a very close ancestor on the call graph with the call sites that call private APIs in the `SpringBoardServices` frameworks. We then manually inspected the functions around the region and found that their least common ancestor on the control-flow graph was a branch that checks if the iOS version was less than 8.0. If so, the application calls the APIs in the `SpringBoardService` framework to get the information about applications on the device; otherwise, it uses the APIs in the `MobileCoreService` framework as

the former ones are blocked. Since our device runs iOS 7.0, such behavior and the additional private APIs would not be revealed had we not used iRiS to analyze the application.

Address	Private API
0xdeec2	SBSSpringBoardServerPort
0xdef46	SBSCopyApplicationDisplayIdentifiers
0xdf056	SBFrontmostApplicationDisplayIdentifier
0xfcf86	IOServiceMatching
0xfcf8e	IOServiceGetMatchingService
0xfd070	IORegistryEntryCreateCFProperty
0xfd0c2	IOObjectRelease
0xfc632	SBSCopyLocalizedApplicationNameForDisplayIdentifier
0xebfaa	[LSApplicationWorkspace defaultWorkspace]
0xebfd0	[LSApplicationWorkspace allApplications]

Table 2: Private API Invocations in APP_S.

The private APIs invoked in APP_S and their call site addresses are listed in Table 2. Since the application collected a lot of user privacy information, we were curious where the information was sent to. To answer this question, we inspected the dynamic execution trace and found that there was a series of API calls right after the private API calls to post a HTTP request to the domain `http://ios.wall.youmi.net`. We then manually reverse engineered the functions along the path in the application and found out the user privacy information was encoded in the URL and sent as part of the HTTP request.

We accessed the domain at `http://www.youmi.net` which is the web site of an advertisement service provider. They provide an advertisement serving library for iOS application developers to use their service, which we suspect might actually collect the user privacy information. The library is provided as binary and headers without source code. To verify our concern, we downloaded the library, built a dummy application with it and analyzed the application using iRiS. As we expected, the application exhibited similar behavior to APP_S and sent user information to this advertisement service provider. It is worth noting that in the advertisement serving library, the Objective-C class names and method names are all obfuscated to random meaningless strings, probably to thwart the effort of manual analysis.

This advertisement service provider claims on their web site that many popular iOS applications have incorporated their advertisement serving library. In fact, in the process of analyzing more iOS applications in our pool, we did find another 20 applications that exhibited similar behavior, which indicates they also use the same library. Compared with individual iOS applications, the existence of such third-party libraries poses greater security risks to user's privacy as they can affect many more users by residing in a large number of applications.

6. LIMITATION

The list of private APIs identified by iRiS might be incomplete since iRiS cannot afford to explore all paths leading to the API call sites in dynamic analysis. In our current implementation, we adopt the linear exploration strategy, which does not reveal private API calls that require a combination of multiple functions to trigger. However, we argue that the problem could be alleviated by using more complex exploration strategies to achieve better path coverage. Large organizations, such as Apple, could provide enough devices to support quadratic or even more complex exploration. We also plan to parallelize our call targets resolving algorithm in future work so multiple devices could be used to speed up the analysis of one application.

iRiS might report private API calls that do not actually happen in real executions since the application might be forced to infeasible paths during the exploration. In such case, we argue that the application should still be considered as suspicious, as it would be very unlikely that a legitimate application happens to have an infeasible path that generates a private API call.

Our current implementation does not cover all types of implicit function invocations in iOS frameworks. For example, the `NSTimer` class allows developers to register a callback function which is called when the timer fires. Handling all such implicit function invocations requires us to thoroughly examine the classes and APIs provided in iOS frameworks, which will be studied in our future work.

iRiS is not able to capture private API calls in control flow generated by external input. Although dynamic code generation is prohibited in iOS, it is still possible to use return oriented programming (ROP) to introduce irregular control flow with external input, as shown in a recent work [30]. Malicious application developers might also choose to use external input, such as network data to create the message selector for Objective-C method calls. In such cases, the control flow could not be determined at the time of application vetting, thus runtime approaches such as control-flow integrity are required to defend against the attack. Nevertheless, we consider our approach to be orthogonal to runtime defense and the two complement each other.

7. RELATED WORK

The work related to iRiS can be classified into three categories: (1) dynamic binary instrumentation, (2) mobile application analysis and (3) mobile runtime hardening.

Dynamic binary instrumentation. Dynamic binary instrumentation frameworks such as PIN [22], Valgrind [23], DynamoRIO [7] and QEMU [6] are widely used for building dynamic analysis systems. All of them work on Android, but none support iOS. Even QEMU, the full system emulator, could not run iOS since it does not emulate the required proprietary hardware used by Apple. In iRiS, we ported Valgrind to iOS to build our dynamic analysis. We envision the availability of dynamic binary instrumentation on iOS will stimulate more future work on iOS security.

Mobile application analysis. There has been a lot of work in Android application analysis. Enck et al. [12] proposed TaintDroid to dynamically track privacy leaks in android applications. Lu et al. [21] presented CHEX which performs static data-flow analysis to detect component hijacking attacks. Zhang et al. [34] presented VetDroid to identify permission use behaviors in android applications using dynamic analysis. Poeplau et al. [26] applied static analysis to detect attempts of loading malicious code in Android applications. Johnson et al. [17] and Wang et al. [31] proposed to switch branch outcomes to expose hidden behavior in Android apps. However, due to the different nature of the two mobile operating systems, it is infeasible to apply these techniques on iOS. For example, most of these analysis systems target the byte code running in the Dalvik VM; in iOS, applications are compiled into native ARM instruc-

tions which are directly executed by the CPU. The access control in iOS is also completely different from the Android permission system.

Compared with Android, little work has been done in the domain of iOS application analysis, which is closely related to iRiS. Egele et al. [11] were the first to present PiOS, a system to analyze privacy leaks in iOS application using static analysis. PiOS uses backward slicing and constant propagation to resolve Objective-C method calls and performs data-flow analysis to identify potential privacy leaks. In iRiS, we use similar approaches in our static analysis stage. Compared with PiOS which only handles the `objc_msgSend` message dispatching function, iRiS covers traditional C function calls, all types of Objective-C message dispatching functions and other implicitly invoked functions, which results in a more complete call graph. Also, as shown in the results of both PiOS and our work, static analysis alone is usually not enough to resolve all call targets in the application binary.

Szydlowski et al. [29] discussed the challenges of performing dynamic analysis on iOS applications. They proposed an approach to identify GUI views in iOS applications using image recognition. The execution of the application is driven by simulating the interaction with identified GUI views using a VNC client. Joorabchi et al. [18] proposed iCrawler to explore the UI states of iOS application by hooking into the application to inspect and exercise the UI elements. Kurtz et al. [20] proposed DiOS which utilizes UI automation to retrieve the GUI hierarchy and interact with GUI elements. All three of these systems adopt the design of driving the execution of an iOS application by interacting with the GUI elements, which suffers from two limitations. First, it is generally infeasible to infer the interaction required to trigger a specific event handler. For example, developers might implement touch event handlers which only recognize and react to specific gestures. Second, even if proper interaction is made on the UI element, the program might refuse to transit to a new UI state when certain conditions are not met. For example, social applications usually require the user to login with his/her account at start. In such cases, the aforementioned systems would get stuck at the login screen and result in a very low code coverage. Contrary to the existing work, iRiS drives the execution of the application by capturing the registration of event handlers and triggers their execution programmatically and applies forced execution so the application can get over various condition checks to reach the desired instructions.

Mobile runtime hardening. In addition to offline mobile application analysis, there also has been work focusing on hardening the execution environment of mobile applications at runtime. Davi et al. [10] proposed MoCFI to enforce control-flow integrity in mobile applications. MoCFI statically rewrites application binaries to add control-flow integrity. Following this work, Werthmann et al. [33] proposed PSiOS which also employs static binary rewriting to add checks that enforce user-defined security and privacy policies. However, both solutions require jailbreaking the end user's iOS device. Recently, Bucicoiu [8] proposed XiOS to prevent use of private APIs in iOS applications. XiOS statically rewrites application binaries to instrument the API call stubs and inserts a reference monitor that checks for private API invocations. XiOS relies on the assumption that all calls to external APIs have to go through the call stubs. However, advanced malicious application developers could scan the address space with the signatures of the target private API functions and obtain the entry addresses to call the private APIs directly, which breaks such an assumption. iRiS detects uses of private APIs in the application vetting stage to complement these runtime defenses.

8. CONCLUSION

With the fast-growing number of third-party iOS applications, the privacy and security of device users becomes an increasing concern. Malicious iOS applications could use private API calls to access sensitive user information. To prevent such attacks, Apple enforces a vetting process for third-party applications to detect the use of private APIs. However, recent attacks have shown that the official vetting process is insufficient to detect advanced forms of private API abuse.

In this paper, we have presented iRiS, an iOS application vetting system that combines static and dynamic analysis to detect uses of private APIs. Since iOS applications are usually large in size, iRiS applies static analysis to resolve the targets of most API invocations. To handle the remaining API invocations that could not be resolved statically, we propose a novel iterative dynamic analysis approach based on forced execution. We port the Valgrind dynamic binary instrumentation framework to iOS to build the dynamic analysis engine for iRiS. To drive the execution of event-driven iOS applications, we propose an automated approach to trigger the execution of the event handlers. Our evaluation with over 2000 iOS applications from an official App Store shows that our technique effectively reveals many uses of private APIs that are not detected by the official vetting process. We found a nontrivial number of applications accessing and sending out sensitive user data, such as installed applications and device serial number. According to our findings, we believe that an advanced application vetting system such as iRiS would be crucial for ensuring the safety of iOS device users.

Acknowledgements

We thank our shepherd, XiaoFeng Wang, and the anonymous reviewers for their constructive comments and suggestions. This work was supported in part by NSF under Award 1409668. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of NSF.

9. REFERENCES

[1] 9to5mac. Former apple employee discusses the app store review process. `http://9to5mac.com/2012/07/04/former-apple-employee-discusses/`.

[2] Apple. Asidentifiermanager class reference. `https://developer.apple.com/library/ios/documentation/AdSupport/Reference/ASIdentifierManager_Ref/`.

[3] Apple. ios developer program license agreement. `http://www.thephoneappcompany.com/ios_program_standard_agreement_20130610.pdf`.

[4] Apple. itunes preview. `https://itunes.apple.com/cn/genre/ios/id36?mt=8`.

[5] Apple. Nib files. `https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/LoadingResources/CocoaNibs/CocoaNibs.html`.

[6] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX ATC'05*.

[7] D. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, 2004.

[8] M. Bucicoiu, L. Davi, R. Deaconescu, and A.-R. Sadeghi. Xios: Extended application sandboxing on ios. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 43–54. ACM, 2015.

[9] BusinessInsider. Apple has shipped 1 billion ios devices. `http://www.businessinsider.com/apple-ships-one-billion-ios-devices-2015-1`.

[10] L. Davi, R. 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 *In Proceedings of the Network and Distributed System Security Symposium (NDSS*, 2012.

[11] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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. *ACM Transactions on Computer Systems (TOCS)*, 2014.

[13] S. Esser. dumpdecrypted. `https://github.com/stefanesser/dumpdecrypted`.

[14] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[15] 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*, pages 272–289. Springer, 2013.

[16] Hex-Rays. Ida pro. `http://www.hex-rays.com/idapro/`.

[17] R. Johnson and A. Stavrou. Forced-path execution for android applications on x86 platforms. In *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*, pages 188–197. IEEE, 2013.

[18] M. E. Joorabchi and A. Mesbah. Reverse engineering ios mobile applications. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 177–186. IEEE, 2012.

[19] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[20] A. Kurtz, A. Weinlein, C. Settgast, and F. Freiling. Dios: Dynamic privacy analysis of ios applications. Technical Report CS-2014-03, Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, June 2014.

[21] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[22] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05*.

[23] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation.

[24] S. Nygard. Class-dump. `http://stevenygard.com/projects/class-dump/`.

[25] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-force: Force-executing binary programs for security applications. In *Proceedings of the 2014 USENIX Security Symposium*, August 2014.

[26] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS'14*.

[27] N. Seriot. ios runtime headers. `https://github.com/nst/iOS-Runtime-Headers`.

[28] Statista. Number of available apps in the apple app store. `http://www.statista.com/statistics/263795/`.

[29] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna. Challenges for dynamic analysis of ios applications. In *Open Problems in Network Security*, pages 65–77. Springer, 2012.

[30] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *Usenix Security*, volume 13, 2013.

[31] Z. Wang, R. Johnson, R. Murmuria, and A. Stavrou. Exposing security risks for commercial mobile devices. In *Computer Network Security*, pages 3–21. Springer, 2012.

[32] R. Watson, W. Morrison, C. Vance, and B. Feldman. The trustedbsd mac framework: Extensible kernel access control for freebsd 5.0. In *USENIX Annual Technical Conference, FREENIX Track*, pages 285–296, 2003.

[33] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz. Psios: bring your own privacy & security to ios devices. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 13–24. ACM, 2013.

[34] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[35] M. Zheng, H. Xue, Y. Zhang, T. Wei, and J. C. Lui. Enpublic apps: Security threats using ios enterprise and developer certificates. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 463–474. ACM, 2015.