PiOS: Detecting Privacy Leaks in iOS Applications

Manuel Egele*†, Christopher Kruegel†, Engin Kirda‡§, and Giovanni Vigna†

* Vienna University of Technology, Austria
manuel@seclab.tuwien.ac.at

† University of California, Santa Barbara
{maeg,chris,vigna}@cs.ucsb.edu

‡ Institute Eurecom, Sophia Antipolis
kirda@eurecom.fr

§ Northeastern University, Boston
ek@ccs.neu.edu

Abstract

With the introduction of Apple's iOS and Google's Android operating systems, the sales of smartphones have exploded. These smartphones have become powerful devices that are basically miniature versions of personal computers. However, the growing popularity and sophistication of smartphones have also increased concerns about the privacy of users who operate these devices. These concerns have been exacerbated by the fact that it has become increasingly easy for users to install and execute third-party applications. To protect its users from malicious applications, Apple has introduced a vetting process. This vetting process should ensure that all applications conform to Apple's (privacy) rules before they can be offered via the App Store. Unfortunately, this vetting process is not well-documented, and there have been cases where malicious applications had to be removed from the App Store after user complaints.

In this paper, we study the privacy threats that applications, written for Apple's iOS, pose to users. To this end, we present a novel approach and a tool, PiOS, that allow us to analyze programs for possible leaks of sensitive information from a mobile device to third parties. PiOS uses static analysis to detect data flows in Mach-0 binaries, compiled from Objective-C code. This is a challenging task due to the way in which Objective-C method calls are implemented. We have analyzed more than 1,400 iPhone applications. Our experiments show that, with the exception of a few bad apples, most applications respect personal identifiable information stored on user's devices. This is even true for applications that are hosted on an unofficial repository (Cydia) and that only run on jailbroken phones. However, we found that more than half of the applications surreptitiously leak the unique ID of the device they are running on. This allows third-parties to create detailed profiles of users' application preferences and usage patterns.

1 Introduction

Mobile phones have rapidly evolved over the last years. The latest generations of smartphones are basically miniature versions of personal computers; they offer not only the possibility to make phone calls and to send messages, but they are a communication and entertainment platform for users to surf the web, send emails, and play games. Mobile phones are also ubiquitous, and allow anywhere, anytime access to information. In the second quarter of 2010 alone, more than 300 million devices were sold worldwide [13].

Given the wide range of applications for mobile phones and their popularity, it is not surprising that these devices store an increasing amount of sensitive information about their users. For example, the address book contains information about the people that a user interacts with. The GPS receiver reveals the exact location of the device. Photos, emails, and the browsing history can all contain private information.

Since the introduction of Apple's iOS[1] and the Android operating systems, smartphone sales have significantly increased. Moreover, the introduction of market places for apps (such as Apple's App Store) has provided a strong economic driving force, and tens of thousands of applications have been developed for iOS and Android. Of course, the ability to run third-party code on a mobile device is a potential security risk. Thus, mechanisms are required to properly protect sensitive data against malicious applications.

Android has a well-defined mediation process that makes the data needs and information accesses transparent to

[1] Apple iOS, formally known as iPhone OS, is the operating system that is running on Apples' iPhone, iPod Touch, and iPad products.

users. With Apple iOS, the situation is different. In principle, there are no technical mechanisms that limit the access that an application has. Instead, users are protected by Apple's developer license agreement [3]. This document defines the acceptable terms for access to sensitive data. An important rule is that an application is prohibited from transmitting any data unless the user expresses her explicit consent. Moreover, an application can ask for permission only when the data is directly required to implement a certain functionality of the application. To enforce the restrictions set out in the license agreement, Apple has introduced a vetting process.

During the vetting process, Apple scrutinizes all applications submitted by third-party developers. If an application is determined to be in compliance with the licencing agreement, it is accepted, digitally signed, and made available through the iTunes App Store. It is important to observe that accessing the App Store is the only way for users with unmodified iOS devices to install applications. This ensures that only Apple-approved programs can run on iPhones (and other Apple products). To be able to install and execute other applications, it is necessary to "jailbreak" the device and disable the check that ensures that only properly signed programs can run.

Unfortunately, the exact details of the vetting process are not known publicly. This makes it difficult to fully trust third-party applications, and it raises doubts about the proper protection of users' data. Moreover, there are known instances (e.g., [20]) in which a malicious application has passed the vetting process, only to be removed from the App Store later when Apple became aware of its offending behavior. For example, in 2009, when Apple realized that the applications created by `Storm8` harvested users phone numbers and other personal information, all applications from this developer were removed from the App Store.

The goal of the work described in this paper is to automatically analyze iOS applications and to study the threat they pose to user data. As a side effect, this also shines some light on the (almost mysterious) vetting process, as we obtain a better understanding of the kinds of information that iOS applications access without asking the user. To analyze iOS applications, we developed PiOS, an automated tool that can identify possible privacy breaches.

PiOS uses static analysis to check applications for the presence of code paths where an application first accesses sensitive information and subsequently transmits this information over the network. Since no source code is available, PiOS has to perform its analysis directly on the binaries. While static, binary analysis is already challenging, the work is further complicated by the fact that most iOS applications are developed in Objective-C.

Objective-C is a superset of the C programming language that extends it with object-oriented features. Typical applications make heavy use of objects, and most function calls are actually object method invocations. Moreover, these method invocations are all funneled through a single dispatch (send message) routine. This makes it difficult to obtain a meaningful program control flow graph (CFG) for a program. However, a CFG is the starting point required for most other interesting program analysis. Thus, we had to develop novel techniques to reconstruct meaningful CFGs for iOS applications. Based on the control flow graphs, we could then perform data flow analysis to identify flows where sensitive data might be leaked without asking for user permission.

Using PiOS, we analyzed 825 free applications available on the iTunes App Store. Moreover, we also examined 582 applications offered through the Cydia repository. The Cydia repository is similar to the App Store in that it offers a collection of iOS applications. However, it is not associated with Apple, and hence, can only be used by jailbroken devices. By checking applications both from the official Apple App Store and Cydia, we can examine whether the risk of privacy leaks increases if unvetted applications are installed.

The contributions of this paper are as follows:

- We present a novel approach that is able to automatically create comprehensive CFGs from binaries compiled from Objective-C code. We can then perform reachability analysis on these CFGs to identify possible leaks of sensitive information from a mobile device to third parties.
- We describe the prototype implementation of our approach, PiOS, that is able to analyze large bodies of iPhone applications, and automatically determines if these applications leak out any private information.
- To show the feasibility of our approach, we have analyzed more than 1,400 iPhone applications. Our results demonstrate that a majority of applications leak the device ID. However, with a few notable exceptions, applications do respect personal identifiable information. This is even true for applications that are not vetted by Apple.

2 System Overview

The goal of PiOS is to detect privacy leaks in applications written for iOS. This makes is necessary to first concretize our notion of a privacy leak. We define as a privacy leak any event in which an iOS application reads sensitive data from the device and sends this data to a third party without the user's consent. To request the user's consent, the application displays a message (via the device's UI) that specifies the data item that should be accessed. Moreover,

the user is given the choice of either granting or denying the access. When an application does not ask for user permission, it is in direct violation of the *iPhone developer program license agreement* [3], which mandates that no sensitive data may be transmitted unless the user has expressed her explicit consent.

The license agreement also states that an application may ask for access permissions only when the proper functionality of the application depends on the availability of the data. Unfortunately, this requirement makes it necessary to understand the semantics of the application and its intended use. Thus, in this paper, we do not consider privacy violations where the user is explicitly asked to grant access to data, but this data is not essential to the program's functionality.

In a next step, we have to decide the types of information that constitute sensitive user data. Turning to the Apple license agreement is of little help. Unfortunately, the text does neither precisely define user data nor enumerate functions that should be considered sensitive. Since the focus of this work is to detect leaks in general, we take a loose approach and consider a wide variety of data that can be accessed through the iOS API as being potentially sensitive. In particular, we used the open-source iOS application Spyphone [17] as inspiration. The purpose of Spyphone is to demonstrate that a significant number of interesting data elements (user and device information) is accessible to programs. Since this is exactly the type of information that we are interested in tracking, we consider these data elements as sensitive. A more detailed overview of sensitive data elements is presented in Section 5.

Data flow analysis. The problem of finding privacy leaks in applications can be framed as a data flow problem. That is, we can find privacy leaks by identifying data flows from input functions that access sensitive data (called *sources*) to functions that transmit this data to third parties (called *sinks*). We also need to check that the user is not asked for permission. Of course, it would be relatively easy to find the location of functions that interact with the user, for example, by displaying a message box. However, it is more challenging to automatically determine whether this interaction actually has the intent of warning the user about the access to sensitive data. In our approach, we use the following heuristic: Whenever there is any user interaction between the point where sensitive information is accessed and the point where this information could be transferred to a third party, we optimistically assume that the purpose of this interaction is to properly warn the user.

As shown in Figure 1, PiOS performs three steps when checking an iOS application for privacy leaks. First, PiOS reconstructs the control flow graph (CFG) of the application. The CFG is the underlying data structure (graph) that is used to find code paths from sensitive sources to sinks. Normally, a CFG is relatively straightforward to extract, even when only the binary code is available. Unfortunately, the situation is different for iOS applications. This is because almost all iOS programs are developed in Objective-C.

Objective-C programs typically make heavy use of objects. As a result, most function calls are actually invocations of instance methods. To make matters worse, these method invocations are all performed through an indirect call of a single dispatch function. Hence, we require novel binary analysis techniques to resolve method invocations, and to determine which piece of code is eventually invoked by the dispatch routine. For this analysis, we first attempt to reconstruct the class hierarchy and inheritance relationships between Objective-C classes. Then, we use backward slicing to identify both the arguments and types of the input parameters to the dispatch routine. This allows us to resolve the actual target of function calls with good accuracy. Based on this information, the control flow graph can be built.

In the second step, PiOS checks the CFG for the presence of paths that connect nodes accessing sensitive information (sources) to nodes interacting with the network (sinks). For this, the system performs a standard reachability analysis.

In the third and final step, PiOS performs data flow analysis along the paths to verify whether sensitive information is indeed flowing from the source to the sink. This requires some special handling for library functions that are not present in the binary, especially those with a variable number of arguments. After the data flow analysis has finished, PiOS reports the source/sink pairs for which it could confirm a data flow. These cases constitute privacy leaks. Moreover, the system also outputs the remaining paths for which no data flow was found. This information is useful to be able to focus manual analysis on a few code paths for which the static analysis might have missed an actual data flow.

3 Background Information

The goal of this section is to provide the reader with the relevant background information about iOS applications, their Mach-O binary format, and the problems that compiled Objective-C code causes for static binary analysis. The details of the PiOS system are then presented in later sections.

3.1 Objective-C

Objective-C is a strict superset of the C programming language that adds object-oriented features to the basic language. Originally developed at NextStep, Apple and its line

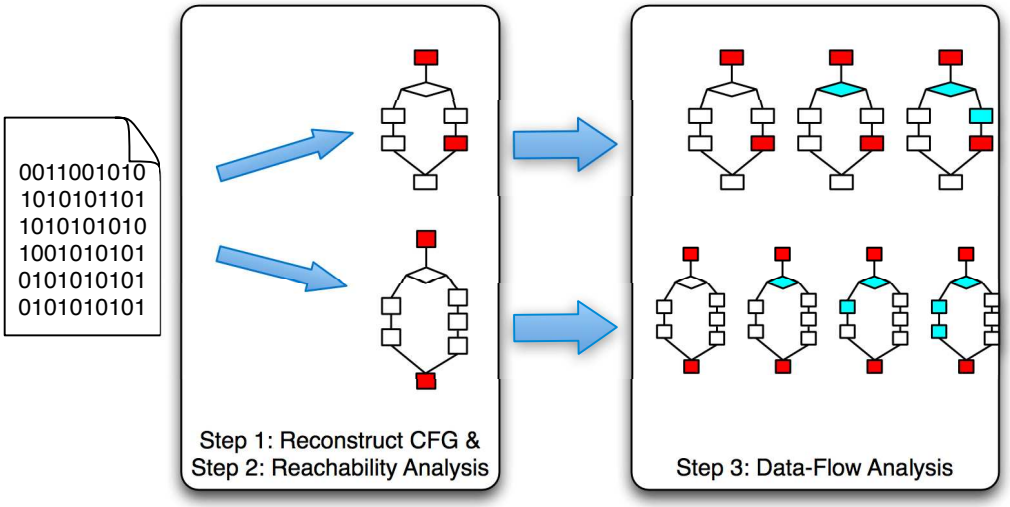

Figure 1. The PiOS system.

of operating systems is now the driving force behind the development of the Objective-C language.

The foundation for the object-oriented aspects in the language is the notion of a class. Objective-C supports single inheritance, where every class has a single superclass. The class hierarchy is rooted at the *NSObject* class. This is the most basic class. Similar to other object-oriented languages, (static) class variables are shared between all instances of the same class. Instance variables, on the other hand, are specific to a single instance. The same holds for class and instance methods.

Protocols and categories. In addition to the features commonly found in object-oriented languages, Objective-C also defines *protocols* and *categories*. Protocols resemble interfaces, and they define sets of optional or mandatory methods. A class is said to adopt a protocol if it implements at least all mandatory methods of the protocol. Protocols themselves do not provide implementations.

Categories resemble aspects, and they are used to extend the capabilities of existing classes by providing the implementations of additional methods. That is, a category allows a developer to extend an existing class with additional functionality, even without access to the source code of the original class.

Message passing. The major difference between Objective-C binaries and binaries compiled from other programming languages (such as C or C++) is that, in Objective-C, objects do not call methods of other objects directly or through virtual method tables (*vtables*). Instead, the interaction between objects is accomplished by sending messages. The delivery of these messages is implemented through a dynamic dispatch function in the Objective-C runtime.

To send a message to a receiver object, a pointer to the *receiver*, the name of the method (the so-called *selector*; a null-terminated string), and the necessary parameters are passed to the `objc_msgSend` runtime function. This function is responsible for dynamically resolving and invoking the method that corresponds to the given selector. To this end, the `objc_msgSend` function traverses the class hierarchy, starting at the receiver object, trying to locate the method that corresponds to the selector. This method can be implemented in either the class itself, or in one of its superclasses. Alternatively, the method can also be part of a category that was previously applied to either the class, or one of its superclasses. If no appropriate method can be found, the runtime returns an "object does not respond to selector" error.

Clearly, finding the proper method to invoke is a non-trivial, dynamic process. This makes it challenging to resolve method calls statically. The process is further complicated by the fact that calls are handled by a dispatch function.

3.2 Mach-O Binary File Format

iOS executables use the Mach-O binary file format, similar to MacOS X. Since many applications for these platforms are developed in Objective-C, the Mach-O format supports specific sections, organized in so-called *commands*, to store additional meta-data about Objective-C programs. For example, the `__objc_classlist` section

contains a list of all classes for which there is an implementation in the binary. These are either classes that the developer has implemented or classes that the static linker has included. The `__objc_classref` section, on the other hand, contains references to all classes that are used by the application. The implementations of these classes need not be contained in the binary itself, but may be provided by the runtime framework (the equivalent of dynamically-linked libraries). It is the responsibility of the dynamic linker to resolve the references in this section when loading the corresponding library. Further sections include information about categories, selectors, or protocols used or referenced by the application.

Apple has been developing the Objective-C runtime as an open-source project. Thus, the specific memory layout of the involved data structures can be found in the header files of the Objective-C runtime. By traversing these structures in the binary (according to the header files), one can reconstruct basic information about the implemented classes. In Section 4.1, we show how we can leverage this information to build a class hierarchy of the analyzed application.

Signatures and encryption. In addition to specific sections that store Objective-C meta-data, the Mach-O file format also supports cryptographic signatures and encrypted binaries. Cryptographic signatures are stored in the `LC_SIGNATURE_INFO` command (part of a section). Upon invoking a signed application, the operating system's loader verifies that the binary has not been modified. This is done by recalculating the signature and matching it against the information stored in the section. If the signatures do not match, the application is terminated.

The `LC_ENCYPTION_INFO` command contains three fields that indicate whether a binary is encrypted and store the offset and the size of the encrypted content. When the field `cryptid` is set, this means that the program is encrypted. In this case, the two remaining fields (`cryptoffset` and `cryptsize`) identify the encrypted region within the binary. When a program is encrypted, the loader tries to retrieve the decryption key from the system's secure key chain. If a key is found, the binary is loaded to memory, and the encrypted region is replaced in memory with an unencrypted version thereof. If no key is found, the application cannot be executed.

3.3 iOS Applications

The mandatory way to install applications on iOS is through Apple's App Store. This store is typically accessed via iTunes. Using iTunes, the requested application bundle is downloaded and stored in a zip archive (with an `.ipa` file extension). This bundle contains the application itself (the binary), data files, such as images, audio tracks, or databases, and meta-data related to the purchase.

All binaries that are available via the App Store are encrypted and digitally signed by Apple. When an application is synchronized onto the mobile device (iPhone, iPad, or iPod), iTunes extracts the application folder from the archive (bundle) and stores it on the device. Furthermore, the decryption key for the application is added to the device's secure key chain. This is required because the application binaries are also stored in encrypted form.

As PiOS requires access to the unencrypted binary code for its analysis, we need to find a way to obtain the decrypted version of a program. Unfortunately, it is not straightforward to extract the application's decryption key from the device (and the operating system's secure key chain). Furthermore, to use these keys, one would also have to implement the proper decryption routines. Thus, we use an alternative method to obtain the decrypted binary code.

Decrypting iOS applications. Apple designed the iPhone platform with the intent to control all software that is executed on the devices. Thus, the design does not intend to give full system (or root) access to a user. Moreover, only signed binaries can be executed. In particular, the loader will not execute a signed binary without a valid signature from Apple. This ensures that only unmodified, Apple-approved applications are executed on the device.

The first step to obtain a decrypted version of an application binary is to lift the restriction that only Apple-approved software can be executed. To this end, one needs to jailbreak the device[2]. The term *jailbreaking* refers to a technique where a flaw in the iOS operating system is exploited to unlock the device, thereby obtaining system-level (root) access. With such elevated privileges, it is possible to modify the system loader so that it accepts any signed binary, even if the signature is not from Apple. That is, the loader will accept any binary as being valid even if it is equipped with a self-signed certificate. Note that jailbroken devices still have access to the iTunes App Store and can download and run Apple-approved applications.

One of the benefits of jailbreaking is that the user obtains immediate access to many development tools ready to be installed on iOS, such as a debugger, a disassembler, and even an SSH server. This makes the second step quite straightforward: The application is launched in the debugger, and a breakpoint is set to the program entry point. Once this breakpoint triggers, we know that the system loader has verified the signature and performed the decryption. Thus, one can dump the memory region that contains the now decrypted code from the address space of the binary.

[2]In July 2010 the Library of Congress which runs the US Copyright Office found that jailbreaking an iPhone is fair use [8].

4 Extracting Control Flow Graphs from Objective-C Binaries

Using the decrypted version of an application binary as input, PiOS first needs to extract the program's interprocedural control flow graph (CFG). Nodes in the CFG are basic blocks. Two nodes connected through an edge indicate a possible flow of control. Basic blocks are continuous instructions with linear control flow. Thus, a basic block is terminated by either a conditional branch, a jump, a call, or the end of a function body.

Disassembly and initial CFG. In an initial step, we need to disassemble the binary. For this, we chose IDA Pro, arguably the most popular disassembler. IDA Pro already has built-in support for the Mach-O binary format, and we implemented our analysis components as plug-ins for the IDA-python interface. Note that while IDA Pro supports the Mach-O binary format, it provides only limited additional support to analyze Objective-C binaries: For example, method names are prepended with the name of the class that implements the method. Similarly, if load or store instructions operate on instance variables, the memory references are annotated accordingly. Unfortunately, IDA Pro does not resolve the actual targets of calls to the `objc_msgSend` dispatch function. It only recognizes the call to the dynamic dispatch function itself. Hence, the resulting CFG is of limited value. The reason is that, to be able to perform a meaningful analysis, it is mandatory to understand which method in which class is invoked whenever a message is sent. That is, PiOS needs to resolve, for every call to the `objc_msgSend` function, what method in what class would be invoked by the dynamic dispatch function during program execution.

Section 4.2 describes how PiOS is able to resolve the targets of calls to the dispatch function. As this process relies on the class hierarchy of a given application, we first discuss how this class hierarchy can be retrieved from an application's binary.

4.1 Building a Class Hierarchy

To reconstruct the class hierarchy of a program, PiOS parses the sections in the Mach-O file that store basic information about the structure of the classes implemented by the binary. The code of Apple's Objective-C runtime is open source, and thus, the exact layout of the involved structures can be retrieved from the corresponding header files. This makes the parsing of the binaries easy.

To start the analysis, the `__objc_classlist` section contains a list of all classes whose implementation is present in the analyzed binary (that is, all classes implemented by the developer or included by the static linker). For each of these classes, we can extract its type and the type of its superclass. Moreover, the entry for each class contains structures that provide additional information, such as the list of implemented methods and the list of class and instance variables. Similarly, the Mach-O binary format mandates sections that describe protocols used in the application, and categories with their implementation details.

In principle, the pointers to the superclasses would be sufficient to recreate the class hierarchy. However, it is important for subsequent analysis steps to also have information about the available methods for each class, as well as the instance and class variables. This information is necessary to answer questions such as "does a class `C`, or any of its superclasses, implement a given method `M`?"

Obviously, not all classes and types used by an application need to be implemented in the binary itself. That is, additional code could be dynamically linked into an application's address space at runtime. Fortunately, as the iOS SDK contains the header files describing the APIs (e.g., classes, methods, protocols, ...) accessible to iOS applications, PiOS can parse these header files and extend the class hierarchy with the additional required information.

4.2 Resolving Method Calls

As mentioned previously, method calls in Objective-C are performed through the dispatch function `objc_msgSend`. This function takes a variable number of arguments (it has a *vararg* prototype). However, the first argument always points to the object that receives the message (that is, the called object), while the second argument holds the selector, a pointer to the name of the method. On the ARM architecture, currently the only architecture supported by iOS, the first two method parameters are passed in the registers R0 and R1, respectively. Additional parameters to the dispatch function, which represent the actual parameters to the method that is invoked, are passed via registers R2, R3, and the stack.

Listing 1 shows a snippet of Objective-C code that initializes a variable of type `NSMutableString` to the string "Hello." This snippet leads to two method invocations (messages). First, a string object is allocated, using the `alloc` method of the `NSMutableString` class. Second, this string object is initialized with the static string "Hello." This is done through the `initWithString` method.

The disassembly in Listing 2 shows that CPU register R0 is initialized with a pointer to the `NSMutableString` class. This is done by first loading the (fixed) address `off_31A0` (instruction: 0x266A) and then dereferencing it (0x266E). Similarly, a pointer to the selector (`alloc`, referenced by address `off_3154`) is loaded into register R1. The addresses of the `NSMutableString` class and the selector refer to elements in the `__objc_classrefs`

and __objc_selrefs sections, respectively. That is, the dynamic linker will patch in the final addresses at runtime. However, since these addresses are fixed (constant) values, they can be directly resolved during static analysis and associated with the proper classes and methods. Once R0 and R1 are set up, the BLX (branch with link exchange) instruction calls the objc_msgSend function in the Objective-C runtime. The result of the alloc method (which is the address of the newly-created string instance) is saved in register R0.

In the next step, the initWithString method is called. This time, the method is not calling a static class function, but an instance method instead. Thus, the address of the receiver of the message is not a static address. In contrast, it is the address that the previous alloc function has returned, and that is already conveniently stored in the correct register (R0). The only thing that is left to do is to load R1 with the proper selector (initWithString) and R2 with a pointer to the static string "Hello" (cfstr_Hello). Again, the BLX instruction calls the objc_msgSend function.

As the example shows, to analyze an Objective-C application, it is necessary to resolve the contents of the involved registers and memory locations when the dispatch function is invoked. To this end, PiOS employs backward slicing to calculate the contents of these registers at every call site to the objc_msgSend function in an application binary. If PiOS is able to determine the type of the receiver (R0) and the value of the selector (R1), it annotates the call site with the specific class and method that will be invoked when the program is executed.

4.2.1 Backward Slicing

To determine the contents of registers R0 and R1 at a call site to the objc_msgSend function, PiOS performs backward slicing [19], starting from those registers. That is, PiOS traverses the binary backwards, recording all instructions that influence or define the values in the target registers. Operands that are referenced in such instructions are resolved recursively. The slicing algorithm terminates if it reaches the start of the function or if all values can be determined statically (i.e., they are statically defined). A value is statically defined if it is a constant operand of an instruction or a static memory location (address).

In Listing 2, for example, the slice for the call to objc_msgSend at address 0x2672 (the alloc call) stops at 0x2668. At this point, the values for both R0 and R1 are statically defined (as the two offsets off_3154 and off_31A0). The slice for the call site at 0x267c (the string initialization) contains the instructions up to 0x2672. The slicing algorithm terminates there because function calls and message send operations store their return values in R0. Thus, R0 is defined to be the result of the message send operation at 0x2668.

Once the slice of instructions influencing the values of R0 and R1 is determined, PiOS performs forward constant propagation. That is, constant values are propagated along the slice according to the semantics of the instructions. For example, MOV operations copy a value from one register to another,[3] and LDR and STR instructions access memory locations.

4.2.2 Tracking Type Information

PiOS does not track (the addresses of) individual instances of classes allocated during runtime. Thus, the question in the previous example is how to handle the return value of the alloc function, which returns a dynamic (and hence, unknown pointer) to a string object. Our key insight is that, for our purposes, the actual address of the string object is not important. Instead, it is only important to know that R0 points to an object of type NSMutableString. Thus, we do not only propagate constants along a slice, but also type information.

In our example, PiOS can determine the return type of the alloc method call at address 0x2672 (the alloc method always returns the same type as its receiver; NSMutableString in this case). This type information is then propagated along the slice. As a result, at address 0x267c, we have at our disposal the crucial information that R0 contains an object of type NSMutableString.

To determine the types of function arguments and return values, our system uses two sources of information. First, for all external methods, the header files specify the precise argument and return types. Unfortunately, there is no such information for the methods implemented in the application binary. More precisely, although the data structure that describes class and instance methods does contain a field that lists the parameter types, the stored information is limited to basic types such as integer, Boolean, or character. All object arguments are defined as a single type *id* and, hence, cannot be distinguished easily.

Therefore, as a second source for type information, PiOS attempts to resolve the precise types of all arguments marked as *id*. To this end, the system examines, for each method, all call sites that invoke this method. For the identified call sites, the system tries to resolve the parameter types by performing the above-mentioned backward slicing and constant propagation steps. Once a parameter type is identified, the meta-data for the method can be updated accordingly. That is, we are building up a database as we learn additional type information for method call arguments.

[3] GCC seems to frequently implement such register transfers as SUB Rd, Rs, #0, or ADD Rd, Rs, #0.

```
NSMutableString *v;
v = [[NSMutableString alloc] initWithString : @''Hello'']
```

Listing 1. Simple Objective-C expression

```
__text:00002668 30 49       LDR     R1, =off_3154
__text:0000266A 31 48       LDR     R0, =off_31A0
__text:0000266C 0C 68       LDR     R4, [R1]
__text:0000266E 00 68       LDR     R0, [R0]
__text:00002670 21 46       MOV     R1, R4
__text:00002672 00 F0 32 E9 BLX     _objc_msgSend   ; NSMutableString alloc
__text:00002676 2F 49       LDR     R1, =off_3190
__text:00002678 2F 4A       LDR     R2, =cfstr_Hello
__text:0000267A 09 68       LDR     R1, [R1]
__text:0000267C 00 F0 2C E9 BLX     _objc_msgSend
                                        ; NSMutableString  initWithString:
```

Listing 2. Disassembly of Listing 1

Frequently, messages are sent to objects that are returned as results of previous method calls. As with method input arguments, precise return type information is only available for functions whose prototypes are defined in header files. However, on the ARM architecture, the return value of a method is always returned in register R0. Thus, for methods that have an implementation in the binary and whose return type is not a basic type, PiOS can derive the return type by determining the type of the value stored in R0 at the end of the called method's body. For this, we again use backward slicing and forward constant propagation. Starting with the last instruction of the method whose return type should be determined, PiOS calculates the slice that defines the type of register R0 at this program location.

4.3 Generating the Control Flow Graph

Once PiOS has determined the type of R0 and the content of R1 at a given call site to `objc_msgSend`, the system checks whether these values are "reasonable." To this end, PiOS verifies that the class hierarchy contains a class that matches the type of R0, and that this class, or any of its superclasses or categories, really implements the method whose name is stored as the selector in R1. Of course, statically determining the necessary values is not always possible. However, note that in cases where only the selector can be determined, PiOS can still reason about the type of the value in R0 if there is exactly one class in the application that implements the selector in question.

When PiOS can resolve the target of a function call through the dispatch routine, this information is leveraged to build the control flow graph of the application. More precisely, when the target of a method call (the recipient of the message) is known, and the implementation of this method is present in the binary under analysis (and not in a dynamic library), PiOS adds an edge from the call site to the target method.

5 Finding Potential Privacy Leaks

The output of the process described in the previous section is an inter-procedural control flow graph of the application under analysis. Based on this graph, we perform reachability analysis to detect privacy leaks. More precisely, we check the graph for the presence of paths from sources (functions that access sensitive data) to sinks (functions that transmit data over the network). In the current implementation of PiOS, we limited the maximum path length to 100 basic blocks.

Interestingly, the way in which iOS implements and handles user interactions implicitly disrupts control flow in the CFG. More precisely, user interface events are reported to the application by sending messages to delegate objects that contain the code to react to these events. These messages are not generated from code the developer wrote, and thus, there is no corresponding edge in our CFG. As a result, when there is a user interaction between the point where a source is accessed, and data is transmitted via a sink, there will never be a path in our CFG. Thus, all paths from sensitive sources to sinks represent potential privacy leaks. Of course, a path from a source to a sink does not necessarily mean that there is an actual data flow. Hence, we perform additional data flow analysis along an interesting path

and attempt to confirm that sensitive information is actually leaked.

5.1 Sources and Sinks

In this section, we discuss in more detail how we identify sources of sensitive data and sinks that could leak this data.

Sources. Sources of sensitive information cover many aspects of the iOS environment. Table 1 enumerates the resources that we consider sensitive. As mentioned previously, this list is based on [17], where Seriot presents a comprehensive list of potentially sensitive information that can be accessed by iOS applications.

Access to the address book
Current GPS coordinates of the device
Unique Device ID
Photo Gallery
Email account information
WiFi connection information
Phone related information (Phone# , last called, etc.)
Youtube application (watched videos and recent search)
MobileSafari settings and history
Keyboard cache

Table 1. Sensitive information sources.

Any iOS application has full read and write access to the address book stored on the device. Access is provided through the `ABAddressBook` API. Thus, whenever an application performs the initial `ABAddressBookCreate` call, we mark this call instruction a source.

An application can only access current GPS coordinates if the user has explicitly granted the application permission to do so. This is enforced by the API, which displays a dialog to the user the first time an application attempts to access the `CoreLocation` functionality. If access is granted, the application can install a delegate with the `CoreLocation` framework that is notified whenever the location is updated by the system. More precisely, the `CoreLocation` framework will invoke the `locationManager:didUpdateToLocation: fromLocation` method of the object that is passed to the `CLLocationManager:setDelegate` method whenever the location is updated.

A unique identifier for the iOS device executing the application is available to all applications through the `UIDevice uniqueIdentifier` method. This ID is represented as a string of 40 hexadecimal characters that uniquely identifies the device.

The keyboard cache is a local file accessible to all applications. This file contains all words that have been typed on the device. The only exception are characters typed into text fields marked to contain passwords.

Furthermore, there exist various property files that provide access to different pieces of sensitive information. The `commcenter` property file contains SIM card serial numbers and IMSI identifiers. The user's phone number can be accessed by querying the `standardUserDefaults` properties. Email account settings are accessible through the `accountsettings` properties file. Similar files exist that contain the history of the Youtube and MobileSafari applications, as well as recent search terms used in these applications. The `wifi` properties file contains the name of wireless networks the device was connected to. Also, a time stamp is stored, and the last time when each connection was active is logged. Accesses related to these properties are all considered sensitive sources by PiOS.

Sinks. We consider sinks as operations that can transmit information over the network, in particular, methods of the `NSURLConnection` class. However, there are also methods in other classes that might result in network requests, and hence, could be used to leak data. For example, the method `initWithContentsOfURL` of the `NSString` class accepts a URL as parameter, fetches the content at that URL, and initializes the string object with this data. To find functions that could leak information, we carefully went through the API documentation. In total, we included 14 sinks.

5.2 Dataflow Analysis

Reachability analysis can only determine that there exists a path in the CFG that connects a source of sensitive information to a sink that performs networking operations. However, these two operations might be unrelated. Thus, to enhance the precision of PiOS, we perform an additional data flow analysis on the paths that the reachability analysis reports. That is, for every path that connects a source and a sink in the CFG, we track the propagation of the information accessed at the source node. If this data reaches one or more method parameters at the sink node, we can confirm a leak of sensitive information, and an alert is raised.

We use a standard data flow analysis that uses forward propagation along the instructions in each path that we have identified. For methods whose implementation (body) is not available in the binary (e.g., external methods such as `initWithString` of the `NSMutableString` class), we conservatively assume that the return value of this function is tainted when one or more one of the arguments is tainted.

Methods with variable number of arguments. To determine whether the output of an external function should

be tainted, we need to inspect all input arguments. This makes functions with a variable number of arguments a little more tricky to handle. The two major types of such functions are string manipulation functions that use a format string (e.g., `NSMutableString appendStringWithFormat`), and initialization functions for aggregate types that fetch the objects to be placed in the aggregate from the stack (e.g., `NSDictionary initWithObjects:andKeys`). Ignoring these functions is not a good option – especially because string manipulation routines are frequently used for processing sensitive data.

For string methods that use format strings, PiOS attempts to determine the concrete value (content) of the format string. If the value can be resolved statically, the number of arguments for this call is determined by counting the number of formatting characters. Hence, PiOS can, during the data flow analysis, taint the output of such a function if any of its arguments is tainted.

The initialization functions fetch the contents for the aggregate from the stack until the value NULL is encountered. Thus, PiOS iteratively tries to statically resolve the values on the stack. If a value statically resolves to NULL, the number of arguments for this call can be determined. However, since it is not guaranteed that the NULL value can be determined statically, we set the upper bound for the number of parameters to 20.

6 Evaluation

We evaluated PiOS on a body of 1,407 applications. 825 are free applications that we obtained from Apple's iTunes store. We downloaded the remaining 582 applications from the popular BigBoss [1] repository which is installed by default with Cydia [12] during jailbreaking. Applications originating from the Cydia repositories are not encrypted. Therefore, these applications can be directly analyzed by PiOS. Applications purchased from the iTunes store, however, need to be decrypted before any binary analysis can be started. Thus, we automated the decryption approach described in Section 3.3.

Since iTunes does not support direct searches for free applications, we rely on apptrakr.com [2] to provide a continuously updated list of popular, free iOS applications. Once a new application is added to their listings, our system automatically downloads the application via iTunes and decrypts it. Subsequently, the application is analyzed with PiOS.

6.1 Resolving Calls to `objc_msgSend`

As part of the static analysis process, PiOS attempts to resolve all calls to the `objc_msgSend` dispatch function. More precisely, for each call to `objc_msgSend`, the system reasons about the target method (and class) that would be invoked during runtime (described in Section 4.2) by the dispatch routine. This is necessary to build the program's control flow graph.

During the course of evaluating PiOS on 1,407 applications, we identified 4,156,612 calls to the message dispatch function. PiOS was able to identify the corresponding class and method for 3,408,421 call sites (82%). Note that PiOS reports success only if the inferred class exists in the class hierarchy, and the selector denotes a method that is implemented by the class, or its ancestors in the hierarchy. These results indicate that a significant portion of the CFGs can be successful reconstructed, despite the binary analysis challenges.

6.2 Advertisement and Tracking Libraries

PiOS resolves all calls to the `objc_msgSend` function regardless of whether the target method in the binary was written by the application developer herself, or whether it is part of a third-party library that was statically linked against the application. In an early stage of our experiments, we realized that many applications contained one (or even multiple instances) of a few popular libraries. Moreover, all these libraries triggered PiOS' privacy leak detection because the system detected paths over which the unique device ID was transmitted to third parties.

A closer examination revealed that most of these libraries are used to display advertisement to users. As many iOS applications include advertisements to create a stream of revenue for the developer, their popularity was not surprising. However, the fact that all these libraries also leak the device IDs of users that install their applications was less expected. Moreover, we also found tracking libraries, whose sole purpose is to collect and compile statistics on application users and usage. Clearly, these libraries send the device ID as a part of their functionality.

Applications that leak device IDs are indeed pervasive, and we found that 656 (or 55% of all applications) in our evaluation data set include either advertisement or tracking libraries. Some applications even include multiple different libraries at once. In fact, these libraries were so frequent that we decided to white-list them; in the sense that it was of no use for PiOS to constantly re-analyze and reconfirm their data flows. More precisely, whenever a path starts from a sensitive sink in a white-listed library, further analysis is skipped for this path. Thus, the analysis results that we report in the subsequent sections only cover the code that was actually written by application developers. For completeness, Table 2 shows how frequently our white-list triggered for different applications.

Library Name	Type	#apps using	#white-listed
AdMob	Advertising	538	55,477
Pinchmedia	Statistics/Tracking	79	2,038
Flurry	Statistics/Tracking	51	386
Mobclix	Advertising	49	1,445
AdWhirl	Advertising	14	319
QWAdView	Advertising	14	219
OMApp	Statistics/Tracking	10	658
ArRoller	Advertising	8	734
AdRollo	Advertising	7	127
MMadView	Advertising	2	96
Total		772	61,499

Table 2. Prevalence of advertising and tracking libraries.

While not directly written by an application developer, libraries that leak device IDs still pose a privacy risk to users. This is because the company that is running the advertisement or statistics service has the possibility to aggregate detailed application usage profiles. In particular, for a popular library, the advertiser could learn precisely which subset of applications (that include this library) are installed on which devices. For example, in our evaluation data set, AdMob is the most-widely-used library to serve advertisements. That is, 82% of the applications that rely on third-party advertising libraries include AdMob. Since each request to the third-party server includes the unique device ID and the application ID, AdMob can easily aggregate which applications are used on any given device.

Obviously, the device ID cannot immediately be linked to a particular user. However, there is always the risk that such a connection can be made by leveraging additional information. For example, AdMob was recently acquired by Google. Hence, if a user happens to have an active Google account and uses her device to access Google's services (e.g., by using GMail), it now becomes possible for Google to tie this user account to a mobile phone device. As a result, the information collected through the ad service can be used to obtain a detailed overview of who is using which applications. Similar considerations apply to many other services (such as social networks like Facebook) that have the potential to link a device ID to a user profile (assuming the user has installed the social networking application).

The aforementioned privacy risk could be mitigated by Apple if an identifier would be used that is unique for the *combination* of application and device. That is, the device ID returned to a program should be different for each application.

6.3 Reachability Analysis

Excluding white-listed accesses to sensitive data, PiOS checked the CFGs of the analyzed applications for the presence of paths that connect sensitive sources to sinks. This analysis resulted in a set of 205 applications that contain at least one path from a source to a sink, and hence, a potential privacy leak. Interestingly, 96 of the 656 applications that triggered the white-list also contain paths in their core application code (i.e., outside of ad or tracking libraries).

The overwhelming majority (i.e., 3,877) of the accessed sources corresponds to the unique device identifier. These accesses originate from 195 distinct applications. 36 applications access the GPS location data at 104 different program locations. Furthermore, PiOS identified 18 paths in 5 applications that start with an access to the address book. One application accesses both the MobileSafari history and the photo storage. An overview that summarizes the potential leaks is shown Table 3.

Source	# App Store	# Cydia	Total
DeviceID	170 (21%)	25 (4%)	195 (14%)
Location	35 (4%)	1 (0.2%)	36 (3%)
Address book	4 (0.5%)	1 (0.2%)	5 (0.4%)
Phone number	1 (0.1%)	0 (0%)	1 (0.1%)
Safari history	0 (0%)	1 (0.2%)	1 (0.1%)
Photos	0 (0%)	1 (0.2%)	1 (0.1%)

Table 3. Applications accessing sensitive data.

An interesting conclusion that one can draw from looking at Table 3 is that, overall, the programs on Cydia are not more aggressive (malicious) than the applications on the App Store. This is somewhat surprising, since Cydia does not implement any vetting process.

6.4 Data Flow Analysis

For the 205 applications that were identified with possible information leaks, PiOS then performed additional analysis to attempt to confirm whether sensitive information is actually leaked. More precisely, the system enumerates all paths in the CFG between a pair of source and sink nodes whose length does not exceed 100 basic blocks. Data flow analysis is then performed on these paths until either a flow indicates that sensitive information is indeed transmitted over the network, or all paths have been analyzed (without result). Note that our analysis is not sound; that is, we might miss data flows due to code constructs that we cannot resolve statically. However, the analysis is precise, and every confirmed flow is indeed a privacy leak. This is use-

ful when the majority of paths actually correspond to leaks, which we found to be true.

For 172 applications, the data flow analysis confirmed a flow of sensitive information to a sink. We manually analyzed the remaining 33 applications to asses whether there really is no data flow, or whether we encountered a false negative. In six applications, even after extensive, manual reverse engineering, we could not find an actual flow. In these cases, our data flow analysis produced the correct result. The remaining 27 cases were missed due to a variety of program constructs that are hard to analyze statically (recall that we operate directly on binary code). We discuss a few of the common problems below.

For six applications, the data flow analysis was unsuccessful because these applications make use of custom-written functions to store data in aggregate types. Also, PiOS does not support nested data structures such as dictionaries stored inside dictionaries.

In four cases, the initial step could not resolve all the necessary object types. For example, PiOS was only able to resolve that the invoked method (the sent message) was `setValue:forHTTPHeader- Field`. However, the object on which the method was called could not be determined. As a result, the analysis could not proceed.

Two applications made use of a JSON library that adds categories to many data types. For example, the `NSDictionary` class is extended with a method that returns the contents of this dictionary as a JSON string. To this end, the method sends each object within the dictionary a `JSONRepresentation` message. The flows of sensitive information were missed because PiOS does not keep track of the object types stored within aggregate data types (e.g., dictionaries).

In other cases, flows were missed due to aliased pointers (two different pointers that refer to the same object), leaks that only occur in the applications exception handler (which PiOS does not support), or a format string that was read from a configuration file.

6.5 Case Studies

When examining the results of our analysis (in Table 3), we can see that most leaks are due to applications that transmit the device ID. This is similar to the situation of the advertising and tracking libraries discussed previously. Moreover, a number of applications transmit the user's location to a third party. These cases, however, cannot be considered real privacy leaks. The reason is that iOS itself warns users (and asks for permission) whenever an application makes use of the `CoreLocation` functionality. Unfortunately, such warnings are not provided when other sensitive data is accessed. In the following, we discuss in more detail the few cases in which the address book, the browser history, and the photo gallery is leaked.

Address book leaks. PiOS indicated a flow of sensitive information for the *Gowalla* social networking application. Closer examination of the offending path showed that the application first accesses the address book and then uses the `loadRequest` method of the `UIWebView` class to launch a web request. As part of this request, the application transmits all user names and their corresponding email addresses.

We then attempted to manually confirm the privacy leak by installing Gowalla on a iOS device and monitoring the network traffic. The names of the methods involved in the leak, `emailsAndNamesQueryString` and `emailsAndNamesFromAddressBook`, both in the `InviterView- Controller` class, made it easy to find the corresponding actions on the user interface. In particular, the aforementioned class is responsible for inviting a user's friends to also download and use the *Gowalla* application. A user can choose to send invitations to her Twitter followers, Facebook friends, or simply select a group of users from the address book. This is certainly legitimate behavior. However, the application also, and before the user makes any selection, transmits the address book in its entirety to the developer. This is the flow that PiOS detects. The resulting message[4] indicates that the developers are using this information to crosscheck with their user database whether any of the user's contacts already use the application. When we discovered this privacy breach, we informed Apple through the "Report a problem" link associated with this application on iTunes. Despite our detailed report, Apple's response indicated that we should discuss our privacy concerns directly with the developer.

PiOS found another leak of address book data in *twitterickl*. This application checks all contacts in the address book to determine whether there is a picture associated with the person. If not, the application attempts to obtain a picture of this person from Facebook. While information from the address book is used to create network requests, these requests are sent to Facebook. It is not the application developers that attempt to harvest address book data.

In other three cases, the address book is also sent without displaying a direct warning to the user before the sensitive data is transferred. However, these applications either clearly inform the user about their activity at the beginning (*Facebook*) or require the user to actively initiate the transfer by selecting contacts from the address book (*XibGameEngine*, to invite friend; *FastAddContacts* to populate the send-to field when opening a mail editor). This shows that not all leaks have the same impact on a user's

[4] "We couldn't find any friends from your Address Book who use Gowalla. Why don't you invite some below?"

privacy, although in all cases, PiOS correctly recognized a sensitive data flow.

Browser history and photo gallery. Mobile-Spy offers an application called *smartphone* on the Cydia market place. This application is advertised as a surveillance solution to monitor children or employees. Running only on jailbroken devices, the software has direct access to SMS messages, emails, GPS coordinates, browser history, and call information. The application is designed as a daemon process running in the background, where it collects all available information and transmits it to Mobile-Spy's site. The user who installs this application can then go to the site and check the collected data.

PiOS was able to detect two flows of sensitive information in this application. The upload of the MobileSafari history, and the upload of the Photo gallery. However, PiOS was not able to identify the leaking of the address book, and the transfer of the email box, or SMS messages. The reason for all three cases is that the application calls `system` with a `cp` command to make a local copy of the local phone databases that hold this information. These copies are later opened, and their content is transferred to the Mobile-Spy service. Tracking through the invocation of the `system` library call would require PiOS to understand the semantics of the passed (shell) commands. Clearly, this is outside of the scope of this paper.

Phone Number. In November 2009, Apple removed all applications developed by Storm8 due to privacy concerns. More precisely, these applications were found to access the user's phone number via the `SBFormattedPhoneNumber` key in the `standardUserDefaults` properties. Once retrieved, the phone number was then transmitted to Storm8's servers. Shortly after the ban of all their applications, Storm8 developers released revised versions that did not contain the offending behavior. This incident prompted Apple to change their vetting process, and now, all applications that access this key are rejected. Thus, to validate PiOS against this known malicious behavior, we obtained a version of Vampires Live (a Storm8 application) that predates this incident, and hence, contains the offending code. PiOS correctly and precisely identified that the phone number is read on program startup and then sent to Storm8.

6.6 Discussion

With the exception of a few bad apples, we found that a significant majority of applications respects the personal user information stored on iOS devices. While this could be taken as a sign that Apple's vetting process is successful, we found similar results for the unchecked programs that are hosted on Cydia, an unofficial repository that can only be accessed with a jailbroken phone. However, the unique device ID of the phone is treated differently, and more than half of the applications leak this information (often because of advertisement and tracking libraries that are bundled with the application). While these IDs cannot be directly linked to a user's identity, they allow third parties to profile user behavior. Moreover, there is always the risk that outside information can be used to eventually make the connection between the device ID and a user.

7 Limitations

Statically determining the receiver and selector for every call to the `objc_msgSend` function is not always possible. Recall that the selector is the name of a method. Typically, this value is a string value stored in the `__objc_selref` section of the application. However, any string value can be converted to a selector, and it is possible to write programs that receive string values whose value cannot be statically determined (e.g., as a response to a networking request, or as a configuration value chosen by the user). This limitation is valid for all static analysis approaches and not specific to PiOS.

Furthermore, aggregate types in Objective-C (e.g., `NSArray`, `NSDictionary`, ...) are not generic. That is, the types of objects in such containers cannot be specified more precisely than `id` (which is of type `NSObject`). For example, the delegate method `touchesEnded:withEvent` of the `UIResponder` class is called whenever the user finishes a touch interaction with the graphical user interface (e.g., click an element, swipe an area, ...). This method receives as the first argument a pointer to an object of type `NSSet`. Although this set solely contains `UITouch` elements, the lack of generic support in Objective-C prohibits the type information to be stored with the aggregate instance. Similarly, any object can be added to an `NSArray`. Thus, PiOS has to treat any value that is retrieved from an aggregate as `NSObject`. Nevertheless, as described in Section 4.2.1, PiOS might still be able to reason about the type of such an object if a subsequent call to the `objc_msgSend` function uses a selector that is implemented by exactly one class.

8 Related Work

Clearly, static analysis and program slicing have been used before. Weiser [19] was the first to formalize a technique called program slicing. As outlined in Section 4.2.1, PiOS makes use of this technique to calculate program slices that define receiver and selector values at call-sites to the `objc_msgSend` dynamic dispatch function.

Also, static binary analysis was used in the past for various purposes. Kruegel et al. [15] made use of static analysis to perform mimicry attacks on advanced intrusion detection systems that monitor system call invocations. Christodorescu and Jha [6] present a static analyzer for executables that is geared towards detecting malicious patterns in binaries even if the content is obfuscated. Similarly, the work described in Christodorescu [7] et al. is also based on binary static analysis, and identifies malicious software using a semantics-aware malware detection algorithm. However, some of the obfuscation techniques available on the x86 architecture cannot be used on ARM based processors. The RISC architecture of ARM facilitates more robust disassembly of binaries, as instructions cannot be nested within other instructions. Furthermore, the strict memory alignment prohibits to jump to the middle of ARM instructions. Thus, disassembling ARM binaries generally produces better results than disassembling x86 binaries.

Note that while static binary analysis is already challenging in any domain, in our work, the analysis is further complicated by the fact that most iOS applications are developed in Objective-C. It is not trivial to obtain a meaningful program control flow graph for iOS applications.

In [4], Calder and Grunwald optimize object code of C++ programs by replacing virtual function calls with direct calls if the program contains exactly one implementation that matches the signature of the virtual function. This is possible because the mangled name of a function stored in an object file, contains information on the class and parameter types. PiOS uses a similar technique to resolve the type of a receiver of a message. However, PiOS only follows this approach if the type of the receiver cannot be determined by backwards slicing and constant propagation.

In another work, Dean et al. [9] present an approach that performs class hierarchy analysis to statically resolve virtual function calls and replace them with direct function calls. In PiOS, we do not use the class hierarchy to resolve the invoked method. However, we do use this information to verify that the results of the backwards slicing and forward propagation step are consistent with the class hierarchy, and thus sensible.

PiOS is also related to existing approaches that perform static data flow analysis. Livshits and Lam [16], for example, use static taint analysis for Java byte-code to identify vulnerabilities that result from incomplete input validation (e.g., SQL injection, cross site scripting). The main focus of Tripp et al. [18] is to make static taint analysis scale to large real-world applications. To this end, the authors introduce hybrid thin-slicing and combine it with taint analysis to analyze large web applications, even if they are based on application frameworks, such as Struts or Spring. Furthermore, Pixy [14] performs inter-procedural, context-sensitive data-flow analysis on PHP web-applications, and also aims to identify such taint-style vulnerabilities.

There has also been some related work in the domain of mobile devices: Enck et al. [10] published TaintDroid, a system that shares a similar goal with this work; namely, the analysis of privacy leaks in smart phone applications. Different to our system, their work targets Android applications and performs *dynamic* information-flow tracking to identify privacy leaks. Most Android applications are executed by the open source Dalvik virtual machine. The information-flow capabilities of TaintDroid were build into a modified version of this VM. iOS applications, in contrast, are compiled into native code and executed by the device's CPU directly. TaintDroid was evaluated on 30 popular Android applications. The results agree quite well with our findings. In particular, many of the advertising and statistics libraries that we identified in Section 6.2 also have corresponding Android versions. As a result, TaintDroid raised alerts when applications transmitted location data to AdMob, Mobclix, and Flurry back-end servers.

Furthermore, Enck et al. [11] present an approach named Kirin where they automatically extract the security manifest of Android applications. Before an application is installed, this manifest is evaluated against so-called *logic invariants*. The result is that the user is only prompted for her consent to install the application if these invariants are violated. That is, only applications that violate a user's assumption of privacy and security are prompted for the user agreement during installation. The concept of a security manifest provides the user basic information on which she can base her decision on whether to install an application or not. Unfortunately, the iOS platform does not provide such amenities. To take a decision, the user can only rely on the verbal description of the application and Apple's application vetting process.

Another work that focuses on Android is the formal language presented by Chaudhuri [5]. Together with operational semantics and a type system, the author created the language with the aim of being able to describe Android applications with regard to security properties. However, the language currently only supports Android-specific constructs. That is, the general Java constructs that build the majority of an application's code cannot currently be represented.

To the best of our knowledge, we are the first to propose an automated approach to perform an in-depth privacy analysis of iOS applications.

9 Conclusions

The growing popularity and sophistication of smartphones, such as the iPhone or devices based on Android, have also increased concerns about the privacy of their

users. To address these concerns, smartphone OS designers have been using different security models to protect the security and privacy of users. For example, Android applications are shipped with a manifest that shows all required permissions to the user at installation time. In contrast, Apple has decided to take the burden off its iPhone users and determine, on their behalf, if an application conforms to the predefined privacy rules. Unfortunately, Apple's vetting process is not public, and there have been cases in the past (e.g., [20]) where vetted applications have been discovered to be violating the privacy rules defined by Apple.

The goal of the work described in this paper is to automatically analyze iOS applications and to study the threat they pose to user data. We present a novel approach that is able to automatically create comprehensive CFGs from binaries compiled from Objective-C code. We can then perform reachability analysis on the generated CFGs and identify private data leaks. We have analyzed more than 1,400 iPhone applications. Our experiments show that most applications do not secretly leak any sensitive information that can be attributed to a person. This is true both for vetted applications on the App Store and those provided by Cydia. However, a majority of applications leaks the device ID, which can provide detailed information about the habits of a user. Moreover, there is always the possibility that additional data is used to tie a device ID to a person, increasing the privacy risks.

Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 257007. This work has also been supported in part by Secure Business Austria and the European Commission through project IST-216026-WOMBAT funded under the 7th framework program. This work was also partially supported by the ONR under grant N000140911042 and by the National Science Foundation (NSF) under grants CNS-0845559, CNS-0905537, and CNS-0716095.

References

[1] `http://thebigboss.org`.

[2] AppTrakr, Complete App Store Ranking. `http://apptrakr.com/`.

[3] iPhone Developer Program License Agreement. `http://www.eff.org/files/20100302_iphone_dev_agr.pdf`.

[4] B. Calder and D. Grunwald. Reducing indirect function call overhead in c++ programs. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 397–408, New York, NY, USA, 1994. ACM.

[5] A. Chaudhuri. Language-based security on android. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009.

[6] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[7] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy (Oakland)*, 2005.

[8] A. Cohen. The iPhone Jailbreak: A Win Against Copyright Creep. `http://www.time.com/time/nation/article/0,8599,2006956,00.html`.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, 1995.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI 2010*, October 2010.

[11] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[12] J. Freeman. `http://cydia.saurik.com/`.

[13] Gartner Newsroom. Competitive Landscape: Mobile Devices, Worldwide, 2Q10. `http://www.gartner.com/it/page.jsp?id=1421013`, 2010.

[14] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[15] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *14th USENIX Security Symposium*, 2005.

[16] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *14th USENIX Security Symposium*, 2005.

[17] N. Seriot. iPhone Privacy. `http://www.blackhat.com/presentations/bh-dc-10/Seriot_Nicolas/BlackHat-DC-2010-Seriot-iPhone%2dPrivacy-slides.pdf`.

[18] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *ACM Conference on Programming Language Design and Implementation*, 2009.

[19] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[20] Wired. Apple Approves, Pulls Flashlight App with Hidden Tethering Mode. `http://www.wired.com/gadgetlab/2010/07/apple-approves-pulls-flashlight%2dapp-with-hidden-tethering-mode/`.