FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Department Informatik

Technical Reports / ISSN 2191-5008

Andreas Kurtz, Andreas Weinlein, Christoph Settgast, Felix Freiling

DiOS: Dynamic Privacy Analysis of iOS Applications

Technical Report CS-2014-03

June 2014

Please cite as:
Andreas Kurtz, Andreas Weinlein, Christoph Settgast, Felix Freiling, “DiOS: Dynamic Privacy Analysis of iOS Applications,” Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, Technical Reports, CS-2014-03, June 2014.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.cs.fau.de

DiOS: Dynamic Privacy Analysis of iOS Applications

Andreas Kurtz, Andreas Weinlein, Christoph Settgast, Felix Freiling
Dept. of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
dios@i1.cs.fau.de

***Abstract*—We present *DiOS*, a practical system to perform automated dynamic privacy analysis of iOS apps. DiOS provides a highly scalable and fully automated solution to schedule apps from the official Apple App Store for privacy analysis to iOS devices. While apps are automatically executed, user interaction is simulated using random and smart execution strategies, and sensitive API calls as well as network connections are tracked. We evaluated the system on 1,136 of the most popular free apps from the iOS App Store and found out that almost 20% of all investigated apps are tracking users' locations on every app start, one third of all accesses to users' address books are attributed to apps from the social network category and almost half of all apps are tracking users' app usage behavior by incorporating tracking and advertising libraries.**

I. Introduction

A. Motivation

Mobile devices have become an integral part of our daily life and business. Not only do they provide connectivity to friends and colleagues at any time and (almost) any place, they are also used to store and manage large amounts of personal data such as images, videos and emails. One major reason for the success of smartphones are the manifold opportunities gained from their expandability by third-party mobile applications (apps).

However, with the increasing amount of apps offered in central marketplaces, the risk of apps that are potentially harmful to a user's privacy rises. Privacy concerns do not necessarily refer only to the secrecy of application data, such as documents and emails, but they also refer to contextual data that a smartphone continuously produces through its sensors for location (GPS), acceleration, video (camera) and audio (microphone). Since most phones are usually switched on and carried along with the person, the collected data gives a deep insight into the life of the smartphone owner, and is therefore highly privacy relevant.

Because of this relevance, the information available on smartphones has attracted more and more attention from the advertising industry and cybercriminals. There have been numerous reports on privacy violations of different forms (even for Apple products [1], [2]), whereas most of them however were discovered by accident or after intensive manual analysis work. Since manual analysis does not scale to the vast amount of apps that are developed every day, automated solutions for detecting privacy leaks are in high demand. Furthermore, since the entire smartphone field is changing rather quickly, it is important to continuously investigate the privacy implications of using smartphones so that users and society can reflect on (self) regulation.

B. Related Work

Related work in dynamic software analysis is mainly focused on desktop operating systems [3]–[5]. Due to several circumstances like, e.g., different execution platforms, most known techniques are not applicable to mobile devices. However, there have been several studies focusing on the analysis of mobile apps, although most of them dealing with Android.

Enck et al. [6] were the first to present TaintDroid, a system for monitoring Android apps for privacy violations at runtime. Later, TaintDroid was incorporated in various other analysis solutions such as AppsPlayground [7] or Mobile-Sandbox [8]. However, TaintDroid (and all other existing solution that are based upon it) are limited to the Android platform and do not support analyzing native code. As code in iOS is directly executed on the hardware without any abstraction layer or virtual machine, those existing solutions are not applicable for iOS-based dynamic analysis. Moreover, compared to the vast amount of Android-related analysis [9]–[13], relatively few studies have been conducted in the field of Apple iOS. This might be attributable to the fact that iOS is a

closed system that provides hardly any interfaces, and information about internals generally need to be reverse engineered.

Szydlowski et al. [14] discussed general challenges of dynamically analyzing iOS apps. Within a prototype implementation, they tracked sensitive API calls using debugger breakpoints and tried to automatically explore an app's user interface (UI) via VNC by recognizing image patterns. In general, the practicability of these approaches is questionable due to performance impacts that are to be expected and, as the authors admit, due to the inaccuracy of this approach to handle non-uniform user interfaces.

Joorabchi and Mesbah [15] propose iCrawler, a tool that explores an app's UI and generates a model of different UI states to facilitate reverse engineering of iOS apps. Although the achieved coverage of their navigation technique looks promising when applied on a few open-source apps, it does not support simulation of any advanced gestures or external events. Moreover, the technique used by iCrawler is only applicable to standard UI elements, and, most notably, iCrawler has not been designed to perform privacy analysis.

Within PiOS, Egele et al. [16] employ data flow analysis and slicing techniques to detect whether an iOS application leaks sensitive data. As of this writing, the results published within the PiOS study are the first and yet only large-scale analysis results on the privacy landscape of iOS apps. However, contrary to our approach PiOS is based on static analysis and prone to shortcomings such as statically resolving message destinations. So overall, there appears to exist no publicly scrutinizable solution for dynamic privacy analysis of iOS apps to date.

C. Contributions

We wish to improve the state of the art in iOS privacy analysis by introducing DiOS. To the best of our knowledge, DiOS is the first fully automated scalable solution to perform *dynamic* privacy analysis of iOS apps.

As apps from the official Apple App Store are compiled for ARM only (and protected by Apple digital rights management), and there exists no emulator for the iOS platform, DiOS schedules apps from Apple's App Store for dynamic privacy analysis to *physical* iOS devices. The analysis throughput scales almost linearly with the number of iPhones added to the system. With only four such devices we were able to dynamically analyze more than 500 apps a day (depending on the desired level of detail).

DiOS allows for structured exploration and navigation of an app's UI by leveraging the automated UI testing support provided by the official Apple development tools. Originally, this feature was intended to simplify UI tests during an app's development phase. However, we successfully reverse engineered the inner workings to retrofit even existing App Store apps to make use of Apple's UI automation features [17]. This allowed us to investigate several robust UI exploration strategies that simulate user interaction and thus optimize an app's UI coverage. We show that our automatic interface navigation achieves comparable high coverage results to manual app usage.

One of the core features of DiOS is its pluggable architecture. While apps are automatically executed and user interaction is simulated using smart execution strategies, any analysis component can be integrated easily. To demonstrate its practicability and reliability, we analyzed 1,136 top free apps from the iOS App Store and tracked privacy-related API calls as well as network connections.

To summarize, this report makes the following contributions:

- We present a fully automated dynamic analysis platform for iOS apps.
- We present a way to automatically purchase apps from the official iOS App Store and to schedule them to physical iOS devices for large-scale privacy analysis.
- We present a novel and robust approach to perform automated UI exploration of iOS apps and explore the merits of different UI execution strategies.
- Finally, we present the results of a large-scale dynamic privacy analysis of 1,136 iOS apps from Apple's App Store.

As we wish to enable other researchers to advance the field of iOS application security based on our framework, the full source code of DiOS, as well as videos demonstrating DiOS in action, are available from our project site: *https://www1.cs.fau.de/dios/*.

D. Roadmap

This report is structured as follows: In Section II, we provide relevant background information on Apple iOS, its basic security concepts and the UI Automation feature. In Section III, we explain the basic architecture of the DiOS analysis framework as well as the challenges we had to overcome to automatically initiate app

purchases and to simulate user interaction. Moreover, we present different execution strategies integrated into DiOS for automated UI exploration. In Section IV, we explain the experimental setup and discuss the results of our large-scale privacy analysis. Finally, we describe the limitations of DiOS and conclude in Section V.

II. Background Information

A. Apple iOS Operating System

iOS (previously iPhone OS) is an operating system used on mobile devices by Apple. iOS is derived from Apple's desktop operating system OS X. While some components of the iOS kernel are open-source, due to historical reasons, the main parts of the iOS operating system are closed and information on it is not publicly available.

The functionality of the iOS operating system can be extended by applications (so-called apps). Those apps can be build using a software development kit (SDK) based on the programming language Objective-C. In order to interact with the underlying operating system, iOS provides an extensive application programming interface (API) organized in several different abstraction layers. While lower layers provide fundamental system services on which all apps rely on, the higher-level layers provide object-oriented abstractions for lower-level constructs.

Most of the iOS system interfaces are provided in special packages, so-called frameworks. A framework consists of a dynamic shared library and corresponding resources (such as header files). Frameworks provide the relevant interfaces needed to build software for the iOS platform. Basically, there are two types of frameworks: *public frameworks* that are recommended by Apple to build third-party apps and *private frameworks* that are restricted to be used by iOS and its system apps only.

B. iOS Security Concepts

The security concept of the iOS system is fundamentally based on its closed nature. Starting from the boot-up process all involved components are cryptographically verified to ensure their integrity and to establish a trusted execution environment. This in turn allows only code providing a valid signature to be executed. Consequently, also third-party apps need to be signed with personalized certificates issued by Apple.

Those third-party apps are usually distributed via the App Store, a digital marketplace maintained by Apple. Although the main purpose of the App Store was to provide users with a service to browse and download apps, it also serves as a unique security feature compared to other more open mobile software platforms like Android. From the beginning, every app has to pass through an approval process by Apple. Among other tests, this vetting process attempts to spot malicious apps and to admit only those apps that are not harmful to users' privacy. Although, this vetting process is a good security design decision at a first glance, there have been reported several cases in which it failed to reveal apps that intentionally accessed personal data [18].

Besides the trusted execution environment and the promised gain in security derived from the App Store approval, there are plenty of other security features provided by the iOS operating system itself [19]. While most of these security features are under the hood and introducing each of them would exceed the scope of this introduction, we only point out the most relevant. First, all third-party apps run isolated within a sandbox and are restricted from accessing other apps' processes and data. If an app needs to access shared data outside its sandbox, it is limited to using the API and services provided by iOS. Access to certain sensitive API methods requires a user's consent. For instance, when an app tries to access a user's current location for the first time, the user is asked for permission once. All privacy-relevant API methods and the related access modalities will be described in Section IV-B in more detail.

C. Jailbreaking

In the past, flaws in the chain of trust and software vulnerabilities in the iOS operating system repeatedly lead to different ways to install custom firmware images and to remove the limitations imposed by Apple. This process is often referred to as *jailbreaking*. One of the main implications of jailbreaking is that it provides root access to the iOS operating system and enables installation and execution of additional software. Signing apps with an Apple-issued certificate is no longer required.

D. Objective-C

As all iOS apps are written in the Objective-C language and an understanding of the basic concepts of Objective-C is required to comprehend certain DiOS principles and limitations, this section outlines the most relevant characteristics of Objective-C.

1) Message Passing: One of the fundamental concepts of the object-oriented programming model of Objective-C is based on message passing. Whenever a method is invoked, a message is sent to an existing object instance. The core of that messaging concept is provided by the `objc_msgSend` function, which is part of the Objective-C runtime environment. This function serves as a central dispatcher and routes messages between existing objects. This dispatcher function takes at least two parameters: The first parameter holds a pointer to the instance of the class to which the message is directed (the *receiver*), while the second argument is the method to be invoked (identified by a *selector*, which is a simple string representing the name of the method). In the end, the dispatcher function may receive a variable list of arguments that are passed to the receiving method implementation.

2) Reflection: Another important characteristic of the Objective-C language is reflection, which is widely known as the ability of a computer program to modify its behavior at runtime. For this, the Objective-C runtime provides a low-level API to interact with the underlying runtime system. By using this API it is possible to query the runtime for information on the running application which gives detailed insights into its inner structure, like available classes, method definitions and instance properties, without the need for having access to the source code. Moreover, this API offers ways to not only derive information on an application's inner workings, but also to manipulate it at runtime. For instance, the runtime provides functions to add new classes and methods, or to rewrite existing method implementations. In the following sections, these characteristics of the Objective-C runtime are the basis for comprehensive and efficient dynamic analysis of iOS applications.

E. API Hooking

In order to intercept calls to the iOS API, an app first needs to be extended with additional code. This can be accomplished by using the environment variable `DYLD_INSERT_LIBRARIES` which advises the dynamic linker to load a specific library into an app's address space at start-up.

During initialization, the injected library needs to reroute the app's control flow using runtime patching techniques. In the case of hooking Objective-C methods, the runtime is instrumented to replace the implementation of specific Objective-C messages. If calls to C functions should be intercepted, the replacement needs to be accomplished at a lower level. For this, it is required to inject jump instructions that reroute an app's control to a dedicated replacement procedure which also resides in the injected library. To not infer with the normal execution flow it is vital to backup the replaced instructions to a custom memory location and to execute them before jumping back to the invoking routine.

In iOS, the most practicable way to hook library calls is by making use of *MobileSubstrate* [20]. MobileSubstrate is a framework that facilitates runtime patches to system functions and consists of two parts: the *MobileLoader* and the *MobileHooker*. First, the MobileLoader integrates itself into an app using the linker environment variable and then loads all dynamic libraries from a specific location. Afterwards, the MobileHooker is responsible for replacing specific system functions and to install the actual hooks.

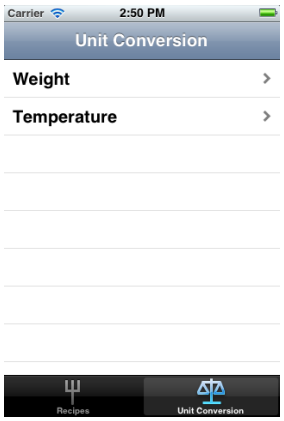
F. UI Automation

With the release of iOS 4, Apple introduced a feature for automated UI testing within its app developer tools. This allows developers to automate UI tests, which are either implemented in JavaScript, or are recorded using a built-in recording function. The resulting test scripts simulate user interaction while an app is running on a connected device using the *UI Automation* API [17], which in turn is based on the iOS *Accessibility* API, a programming interface to make an app's elements such as buttons, input fields etc. usable to visually impaired people.

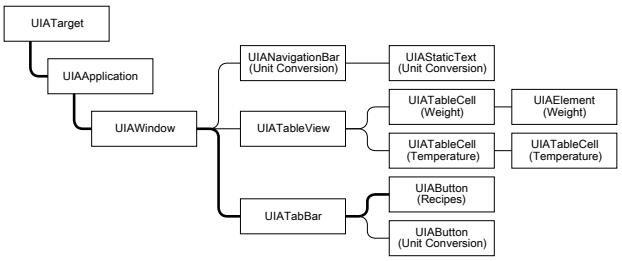
```
1 var localTarget = UITarget.localTarget
      ();
2 localTarget.frontMostApp().mainWindow.
      tabBar().buttons()["Recipes"].tap();
3 localTarget.frontMostApp().mainWindow.
      tabBar().buttons()[0].tap();
```

Listing 1: UI Automation test script to automatically tap the recipes button in Figure 1

To understand how automatic exploration of an user interface works, it is first necessary to understand an app's internal user interface structure. Therefore, figure 1a shows a sample app consisting of a navigation bar at the top, a table view to display the actual contents, and a tab bar at the bottom edge of the screen to change content selection by tapping one of the two buttons. All of these elements are internally represented within a hierarchy as illustrated in Figure 1b. At the top of the element hierarchy is the `UITarget` class which



(a) Sample recipes app



(b) Hierarchy of the user interface elements displayed in Figure 1a

Fig. 1: Basic user interface of a sample app and its internal representation of the element hierarchy [21].

is a high-level representation of the iOS device. The running app is the *frontmost app* and can be accessed, as well as any other user interface elements, from an UI Automation test script as shown in Listing 1. For example, in order to tap the recipes button, the path to the selected node has to be stated. Array elements can either be accessed by name or by index.

One limitation of UI Automation is that access to the element hierarchy is only available for standard UI elements or custom elements that have explicitly been made accessible using the Accessibility API. With the goal of automated app exploration in mind, this drawback can be neglected for several reasons: Most of the apps available in the App Store comply with the *iOS Human Interface Guidelines* [22] by Apple, which encourage developers to only build apps consistent with iOS standards. This includes a uniform style and results in a preferred use of standard UI elements. If an app is not offering accessibility support, as it is e.g. often the case with games, information on the displayed elements cannot be extracted. However, UI Automation can still be used, as it allows simulation of several gestures at any specific point on the screen without knowing the actual element hierarchy. Apart from touch gestures, UI Automation has also the ability to simulate external events like changes to the location, device orientation, volume etc. Moreover, UI Automation provides a way to handle alert messages to ensure that alerts do not interfere while an app is automatically executed.

As described in more detail later, the automatic UI exploration within DiOS is heavily based on this UI Automation feature. Although intended to be used during app development and testing only, we successfully reverse engineered the UI Automation components to retrofit even existing App Store app's to make use of it. This provides us with a clean interface to explore an app's internal element hierarchy, which in fact is represented by an element tree, and it even allows us to simulate user interaction in a convenient and robust way.

III. DiOS Framework

In the following, we present the design of DiOS (Section III-A) and describe how we met some of the major challenges within iOS-based dynamic analysis, such as to automatically initiate app purchases from Apple's App Store (Section III-B), to simulate user interaction to connected devices (Section III-C) or to deal with alert messages (Section III-D). Afterwards, we present different execution strategies integrated into DiOS for automated UI exploration (Section III-E) and evaluate the framework's scalability and performance (Section III-F).

A. System Architecture

Basically, the DiOS system consists of three major parts as illustrated in Figure 2: a backend that is mainly used as central data storage, a worker used as connecting link between the backend and any number of attached iOS devices, and several client components running on the connected iOS devices. Each of these components will be explained in more detail within the following sections.

1) Backend: The *Backend* is the main component of DiOS and is responsible for centralized data storage. It holds configuration data such as App Store account credentials (to automatically purchase apps) and status information on available iOS analysis devices. Furthermore, it archives all app binaries of recent analysis and processes analysis results reported back from the iOS devices. To browse the App Store, DiOS provides a web-based interface that uses RSS feeds provided by Apple [23] to retrieve information on available apps and

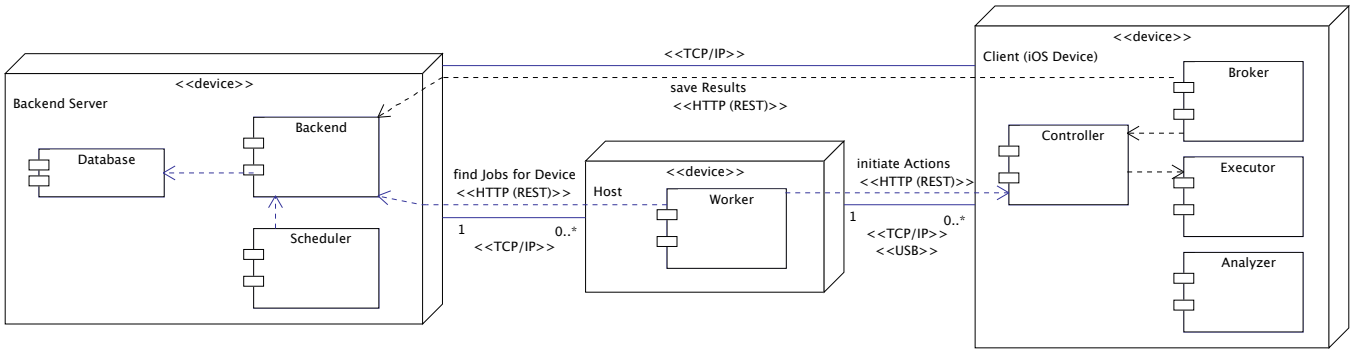


Fig. 2: Architectural overview of the DiOS framework. Components on the right-hand side are executed on attached iOS devices.

app charts. A scheduler ensures that selected apps are automatically installed and executed on available iOS devices. The backend component was implemented in Python and provides a web service interface to enable any other DiOS component to access its data.

2) Worker: The main task of the *Worker* is to establish a physical link between all available iOS devices (that are connected to the worker via USB) and the backend. This intermediate layer was mainly introduced for scalability reasons, as DiOS supports multiple worker instances. Whenever a new device is connected to a worker for the first time via USB, it is automatically set up and registered with the backend. Afterwards, the worker constantly polls the backend for pending jobs and, on demand, initiates app installations and app launches on the connected iOS devices. The worker component was implemented in Python. To facilitate communication via USB the worker uses *libimobiledevice* [24], a software protocol library to natively communicate to iOS devices.

3) Client: As the DiOS model is based on executing and analyzing running apps on connected iOS devices, several client components are required. These client components running on the iOS devices are explained in more detail in the following.

- The *Controller* is the frontmost client component that receives instructions from the worker such as purchasing apps from the App Store, installing and executing apps and delegating automatic UI exploration to the executor component. The controller is implemented in Objective-C and is integrated into SpringBoard (the standard application that manages the iOS home screen) via library injection. During initialization of the controller library an HTTP web server is spawned providing a REST-based web service interface to the worker component. Further, on-device communication between all involved client components is realized using the distributed notification concept of iOS [25].
- The *Executor* is responsible for exploring an app's user interface and for simulating user interaction. For this, several execution strategies were implemented which provide different granularity levels of UI exploration (Section III-E). To separate automation code from the actual app code, the executor was implemented as iOS background daemon and makes use of the UI Automation API [17] (Section III-C).
- The *Analyzer* is the core of DiOS' plugin architecture. All available analyzer components are loaded into an app's address space at startup and are responsible for performing the actual app analysis. By default, DiOS provides a basic analyzer that tracks access to privacy-related API calls and collects network packages.
- The *Broker* is a static library that simplifies the development of DiOS analyzer plugins. It receives notifications from the controller and provides a clean interface to report analysis results to the backend.

B. Automated App Purchase and Installation

In order to enable large-scale analysis of iOS apps, the process of automatically purchasing apps from the App Store and downloading them to connected iOS devices is a key part of the DiOS client components. However, so far app purchasing is limited to official apps provided by Apple only and not available to any third-party implementations. Therefore, we reverse

engineered the official iOS App Store app to find out how to initiate store purchases from the DiOS controller component in an automatic fashion.

Thereby, it turned out that the private *StoreServices* framework is used to purchase any available store products. In more detail, first a `SSPurchase` object is initialized. This object is supplied with information on the related store account to be used for the purchase and several additional buying parameters[1] which include the bundle identifier of the actual app. Finally, a `SSPurchaseRequest` object is initialized to deliver the purchase request object to the App Store and to start the download process.

In this context, a major challenge was to properly handle authentication forms, as a purchase requires a user to enter his or her App Store credentials. We found out that these credentials were cached in memory and could be re-used without any additional user confirmation in iOS version 5 and below. Thus, it was sufficient to set the property `setNeedsAuthentication` to `false` before submitting the request. However, this behavior has changed with the release of iOS version 6 where cached credentials now expire at irregular intervals and therefore require to be re-entered. Therefore, we instrumented the alert handling system of DiOS (see Section III-D) to automatically submit store credentials to enable fully automated purchases.

C. Co-Opting the UI Automation Service

As mentioned above, Apple's UI Automation feature was intended to support automated UI testing during development. From within the Apple developer tools, JavaScript-based test scripts can be recorded in order to automatically navigate an app through previously defined paths. In order to retrofit even existing App Store apps to make use of Apple's UI automation features, we reverse engineered its inner workings.

We found a *script agent* running on the iOS device responsible for parsing the received test scripts and for translating them into Objective-C method invocations. The related Objective-C methods are provided within the private `UIAutomation.framework`, an on-device counterpart of the publicly available JavaScript API [17]. This private framework is not part of the operating system itself and needs to be extracted from the iOS developer image, a toolset which is restricted to iOS developers and loaded onto an iOS device on-demand. The reverse engineering of the UI Automation framework was facilitated by the fact that its method structure is almost equivalent to its official JavaScript counterpart.

After reconstructing the internal Objective-C API, we were able to leverage the UI Automation framework to automatically inspect an app's user interface at runtime and to simulate any user interaction or trigger external events, without requiring any developer tools or an app's source code.

D. Dealing with Alerts

iOS provides different kinds of alerts in order to display notifications or to ask users for basic decisions. As these alerts would interrupt the automatic execution of an app, proper alert handling was also a notable challenge. To solve this, the DiOS Executor (see Section III-A) was extended by an *AlertManager*. This AlertManager is registered as callback via the UIAutomation framework and is automatically invoked whenever an alert rises. For maximum flexibility, the AlertManager delegates further processing of an alert to several *AlertHandler*s depending on an alert's origin (system vs. app-specific).

In consequence, system alerts or unsolved app-specific alerts are processed by a *GenericAlertHandler*. Usually, this handler results in pressing an alert's default button. However, a few special cases have emerged that require special treatment. For instance, whenever an app tries to access certain privacy-related API methods that requires a user's consent, the GenericAlertHandler explicitly grants access by pressing the *"allow"* button. Another special case was observed within apps that ask users at startup to rate their app experiences or to download affiliated apps. Pressing the default button would forward the user to the App Store app, while redirecting back to the original app may result in an infinite loop, pending execution between these two apps. To avoid those behaviors, we implemented a blacklist of button names (like *rate, evaluate, try* etc.) that are to be avoided.

In contrast, app-specific alerts are treated individually depending on the the selected execution strategy. These strategies will be explained within the following sections in more detail.

[1]Information required to automatically fill in an app's buying parameters can be obtained by scraping its HTML description from the App Store when `iTunes-iPhone` is provided within the HTTP user agent request header.

E. Execution Strategies

To automatically navigate through an app's user interface and to simulate user interaction, the following three different execution strategies have been integrated into DiOS.

1) Open-Close Execution: The most basic execution strategy within DiOS launches an app for a defined duration. After a few seconds (15*s* by default) the app is suspended to the background by simulating a tap on the iOS device's home button. Again, five seconds later the app is resumed for another 15 seconds until execution is finally terminated. Apart from handling alerts by pressing the default button no other user interaction is simulated. Therefore, the results of this strategy later serves as a baseline reference to measure the advance of the competing approaches.

2) Random Execution: Moreover, DiOS provides a random execution strategy that utilizes Apple's UI Automation framework to execute randomly chosen events, from pressing the home-button, changing a devices orientation or location, shaking the device, or adjusting the volume, to tapping on a random screen position or performing any advanced gestures. However, compared to common Android-based UI automation approaches [7], [8], DiOS does not rely on issuing random events only, but considers an app's actual user interface in certain ways: If an app provides any navigation bars, from time to time, a random button of these navigation elements is pressed intentionally to avoid the risk of being stuck to sub views. Moreover, to allow the simulated user interaction to be as realistic as possible, the random events were weighted and can be freely adjusted. Thus, the event of tapping a random screen position, e.g., is ten times more likely to occur than changing a device's orientation. Using this approach we aimed to effectively maximize an app's coverage without the need for exploring the actual UI in detail.

3) Smart Execution: The third execution strategy implemented within DiOS examines the actual user interface in great detail. For this, it explores the available UI elements on every screen and keeps track of already executed UI paths. This enables targeted navigation through an app's user interface, while avoiding repeated execution of already visited contents.

In order to (re-)recognize different UI states of an app, it is necessary to define criteria to determine the similarity of two *views*. Thereby, a view is defined as an abstract description of an ordered set of different UI elements (buttons, textfields etc.) that are arranged on the screen in a certain way. Within DiOS we define two views as being equivalent if and only if the following conditions are met:

1) The element hierarchy is identical.
2) The following identifiers of each hierarchy element are identical: UIAutomation class name, UIKit class name, UIAutomation class name of the parent element.

In practice, this abstraction provides a reliable way to determine UI equivalence within iOS and to differentiate UI states without considering any variable UI elements (like label/button names, values). To keep track of all available app views, we modeled the different UI states as nodes within a directed graph. The related UI elements that lead to specific UI states are represented by edges within that graph. This model allows a targeted and efficient navigation between different views. Whenever all available elements of a view have been examined and successfully executed, the model is searched for pending elements in other UI states. Finally, we calculate the shortest path to efficiently navigate to that view.

In more detail, the following steps are performed whenever a view is entered. At the beginning, the element hierarchy of this view is retrieved using the UIAutomation framework. Afterwards, the state model graph is searched for the current view using the UI equivalence criteria described above. In case the current view is not available within the state model, it is added to the graph. Subsequently, a non-processed element is randomly chosen from the current view, any available input fields are filled using default data and the element is executed.

One shortcoming of the smart execution approach is the time needed to perform the numerous calculations at the beginning of each transaction. In practice, these calculations require approximately five seconds (on iPhone 4 hardware). In exceptional cases, especially within complex user interfaces, the iterations may take longer. There are several reasons for this: On the one hand, complex views may contain a large number of (hidden) elements and processing these elements takes some time, mainly caused by architectural issues. On the other hand, the time delay is increased by long response times of the UIAutomation framework to provide information on the element hierarchy. We addressed these issues by implementing concurrency control and shifting the element analysis into separate dispatch queues.

F. Framework Evaluation

The main objective of DiOS was to provide a system for large-scale privacy analysis of iOS apps from the official Apple App Store. Within this section we evaluate and discuss the scalability of the overall system as well as the effectiveness of the three different execution strategies. Therefore, we analyzed the top 50 free apps of each of the 23 available App Store categories (as of May 10, 2013). We downloaded these 1,136 apps (duplicate entries were removed) and analyzed them using different execution strategies.

1) Scalability: The architecture of DiOS was chosen to be as scalable as possible. This is particularly given by the loose coupling of the involved components. While DiOS is based on executing apps on physical devices (iPhones), the analysis throughput scales almost linearly with the number of iPhones added to the system.

During our experiments, the required time to download[2] an averaged size app, to install it on a connected device and to remove it after successful execution was around three to four minutes. Given that constant time lapse of 3.5 minutes, the overall analysis time per app solely depends on the chosen execution time. Thus, running an app for five minutes, e.g., using smart execution strategies, the overall analysis takes 8.5 minutes to complete. This results in a throughput of seven apps per hour, or 169 apps per day. In order to increase the throughput, DiOS allows to add any number of iOS devices. Thus, to analyze 1,000 apps per day which is approximately the number of submitted apps to the App Store [26], only 6 analysis devices are required.

2) Method Coverage: One major drawback of dynamic analysis in general is that its detection abilities highly depend on the achieved code coverage. The code coverage again is affected by proper simulation of UI events, particularly in the context of highly interactive mobile apps. If an app performs sensitive API calls within a sub-branch that requires specific UI interaction to be executed this event might be missed, increasing the risk of false negatives. This is why accurate simulation of user interaction is a key part of any UI-driven dynamic analysis system.

In order to compare the different execution strategies provided by DiOS, we instrumented the Objective-C runtime to track all method invocations and for each execution strategy measured the amount of invoked app methods. Although method coverage is known to be a weak coverage criteria in systematic software testing, it has proven to be a suitable benchmark to check DiOS' execution strategies against each other. Compared to other platforms like Android where a mid-layer (Dalvik VM) can be instrumented to measure the coverage, iOS apps are executed on the hardware directly. Measuring the coverage on a basic block or instruction level might produce more significant results but is considered to be impractical within large-scale analysis due to performance impacts.

a) Tracing Method Invocations: As noted above, every method invocation in Objective-C results in one or more messages to the `objc_msgSend` runtime function. By utilizing the Objective-C runtime we were able to intercept all messages to that central dispatcher and to retrieve a clear trace of all method invocations. In detail, we replaced the implementation of the `objc_msgSend` function in order to log the CPU registers `R0` and `R1` which hold a pointer to the receiving class instance and the name of the calling method at this stage. As the logging procedure itself modifies the CPU registers as well as the stack, it is essential to preserve the current execution state within the prologue of the replacement function. On assembly level, we allocated an alternate stack within heap memory and copied the values of the main CPU registers including the *Link Register* to this alternate stack. Following this, we logged the related register values into a database, resolved the associated class name of the receiving instance and restored the main stack before continuing normal execution. To focus on app-specific method invocations only, we enumerated all classes and methods of each running app using the Objective-C runtime API. Using this approach we were able to precisely determine the amount of invoked Objective-C app methods.

b) Comparison of Execution Strategies: We evaluated the achieved method coverage of each execution strategy within several experiments. Therefore, we executed each of the above mentioned 1,136 App Store apps using the open-close, the random and the smart execution strategy. In each case, the execution time was limited to three minutes, except one additional test run using the smart execution strategy for another five minutes.

The experimental results are shown in Figure 3a. While starting an app without any further user interaction (Open-Close Execution), 13 % of all available app

[2]The time calculations are based on an app's average file size of 20 MB, downloaded using a high-speed internet connection of > 500KB/s.

methods were accessed. By simulating random events (Random Execution) for three minutes, the average coverage rate was increased to 17 %. Compared to that, three minutes of running an app using Smart Execution achieved an average method coverage of only 16 %, while executing an app for five minutes using Smart Execution further increased the coverage to 18 %. When comparing the results by category, we observed various differences. While some apps like, e.g., Weather apps achieve a higher coverage due to the synchronization of weather data during start up, social networking apps often could not progress past the initial welcome screen due to missing login credentials.

It is notable that within a three minute execution time frame, simulating random events achieves higher coverages than driving an app's user interface smartly. There are several reasons for this: First, random execution already takes an app's UI state into account as the events are weighted and it preferably tabs navigation bar buttons. Therefore, it should be considered rather a hybrid of random and smart execution. Second, Smart Execution takes a long time to examine an app's user interface via UI Automation and to maintain the UI state model. This is why we supposed a strong dependence on the execution time. In order to study the actual impact of the execution time, we selected 67 apps from our sample set and executed them for additional 30 minutes. The selection was mainly based on those apps at which the coverage has improved by 5 % within five minutes of Smart Execution. By running these selected apps for additional 30 minutes using Smart Execution, we observed an average method coverage of 23 %. This represents an average increase of 9 % per app compared to the initial Open-Close Execution (see Figure 3b).

Although we observed an increase in coverage by raising the execution time, we still found the overall coverage relatively low (< 30 %). This is why we investigated the actual amount of invoked app methods when an app is used manually. Therefore, we randomly selected 23 apps (one of each category) and executed each of them by hand for five minutes. We tried to cover as much UI elements as possible, while tracking the amount of invoked app methods. We observed an average method coverage of around 27.3 % after three minutes and 29.4 % after five minutes. Our results indicate that even within ordinary app usage, the achieved code coverage reaches a saturation point at around 30 %. We noticed that most of the idle methods are provided by view controllers, libraries and obsolete testing routines which are either invoked within certain conditions only or not used at all. At least, it was not possible to invoke these methods by browsing the user interface for five minutes by hand. These results underline the strengths of our automatic UI exploration techniques that achieve comparable high coverage results.

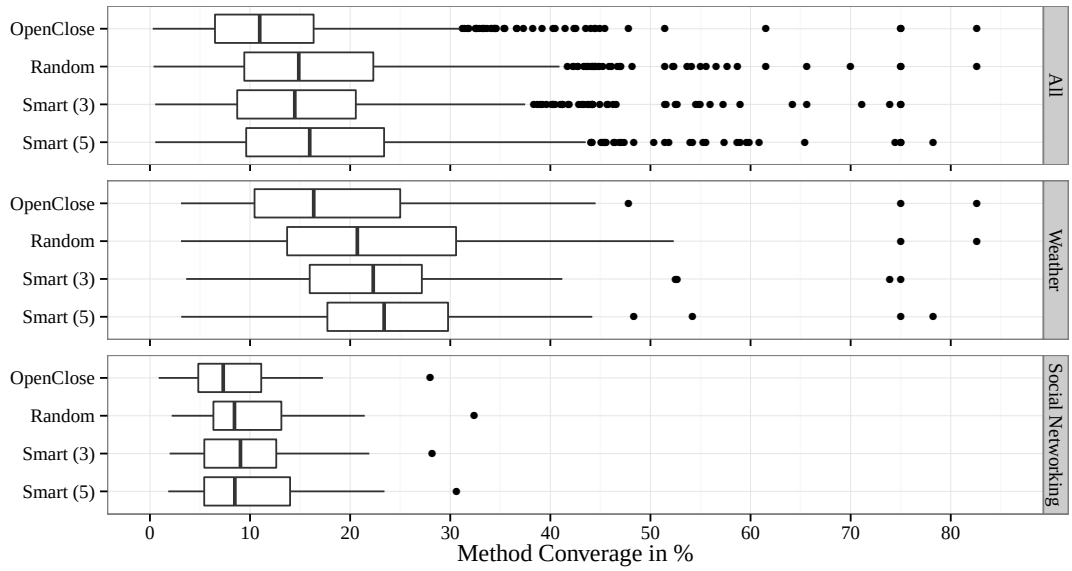
To date, only a few studies have examined the amount of covered code achieved through automatically browsing a mobile app's user interface. Szydlowski et al. [14] observed a coverage rate of 16 % during the startup phase of an iOS app. These results are mostly consistent with our observations (∼13 %). Gilbert et al. [10] executed nine Android apps using a random simulation approach. The authors observed a basic block code coverage of 40% or lower and also experienced the limitations of UI exploration regarding social networking apps, where they measured a coverage of less than 1%. Finally, the authors of the Android analysis platform TraceDroid [27] observed a maximum coverage rate of 33.6 % when applying their solution to 35 Android apps. Moreover, they found the overall coverage of stimulating apps manually at a comparable low level (32.9 %), which again confirms our iOS-based observations. Altogether, the observations outlined in this report provides for the first time key insights into the amount of covered code by automatically exploring the user interfaces of a fairly large quantity of *real* iOS App Store apps.

IV. Large-Scale Privacy Analysis of iOS Apps

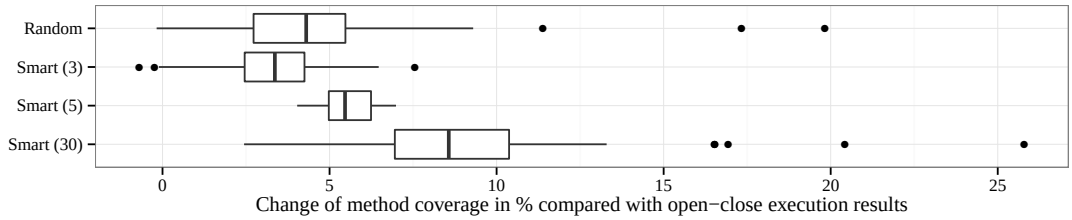
In order to demonstrate the scalability and reliability of DiOS we analyzed our selection of 1,136 top free apps (as of May 10, 2013) for privacy issues. The primary goal was to give an overview of privacy-related API accesses and network communication. Within this section, we describe the experimental setup, the privacy criteria our study was based and the results we observed.

A. Experimental Setup

Our analysis was carried out using four iPhone 4 devices running iOS 6 which have been prepared with a basic DiOS analyzer component that tracks an app's access to privacy-related API methods and its network communication. Within a first experiment we executed each app using DiOS' basic Open-Close execution strategy to establish a baseline reference of privacy-related data that is accessed by only downloading and starting an app from Apple's App Store. Within a second



(a) Amount of invoked app methods by execution strategy grouped by selected categories



(b) Change of method coverage by comparing several execution strategies to basic Open-Close Execution

Fig. 3: Comparison of achieved method coverage by executing apps automatically using different execution strategies.

run using DiOS' Smart Execution, we determined any deviation from our baseline reference when apps are executed and automatically navigated for five minutes.

B. Analysis Criteria

Within our analysis we were interested in tracking an app's access to all personal data available on a mobile device such as a user's contacts, calendars, photos, and hardware sensors like the microphone, camera, accelerometer, GPS etc. Therefore, we extracted all relevant API methods and monitored each access. Although some of these methods require a user's consent to be invoked, we were interested in tracking them anyway, as excessive access to these data may raise downstream privacy issues. Moreover, we compiled a list of the most popular advertising and analytic providers for iOS and hooked each specific initialization methods. As most of those services permanently track a user's app usage behavior, we consider them highly privacy-relevant.

In the end, one major advantage of dynamic analysis over static analysis [16] is the ability to easily track runtime information like network traffic. Thus, we also monitored the related iOS methods that are used for HTTP communication.

While implementing our analysis library, we faced several challenges of determining the real source of a specific API access. For instance, system frameworks often invoke sensitive API within normal operation. Those regular accesses need to be separated from method calls originating from an app itself, in order to avoid false positives. Therefore, we utilized the Objective-C runtime to return a backtrace of the current call stack every time a privacy-related method was invoked. Thus, we were able to determine the names of the calling methods, to identify the inquiring code part and to ignore accesses from system frameworks.

C. Results

The following presentation of our analysis results is divided into five parts.

1) Personal Data: For obvious reasons, invocations of API methods providing direct access to personal data are the most relevant parameters to study privacy implications. Unsurprisingly, we found a user's calendar most frequently accessed from apps of the *productivity* (30 %) and *business* (20 %) categories, and most accesses to a user's photos for apps of the *photos & video* category (35.4 %). However, it is important to highlight that almost one third of all accesses to a user's contacts are attributed to apps from the *social network* (29 %) category, followed by *travel* (19.4 %) and *navigation* (9.7 %) apps.

Moreover, we revealed that almost 20 % of all investigated apps access a user's location at every startup. After five minutes of execution, the average amount increased to 25.6 % (see Figure 4). As expected, most of the location accesses were registered within apps from the *navigation* (15.5 %), followed by apps from the *weather* (14 %) category, obviously to display contextual data. One in five *social networking* apps also accesses a user's location on every app start. A prominent example is the official *Facebook* app that collects a user's position on every launch, without any obvious usage scenario. This is especially precarious as most phones are usually switched on and carried along and, therefore, the collected location data gives a deep insight into users' habits and personal activities [28].

Finally, accesses to a device's camera, microphone and other sensors like the accelerometer are of particular interest, as sensor access is unrestricted within iOS 6 and does not require a user's consent. While accesses to the microphone and the camera, as depicted in Figure 4, are privacy-relevant for obvious reasons, risks associated with the accelerometer sensor need further explanation. Marquardt et al. [29] demonstrated that the data provided by the accelerometer sensor is precise enough, to allow an app to recover text entered on a nearby keyboard. Though we do not assume that this practice is widespread, however, 15.7 % of all apps would be able to make use of this technique.

2) Advertising and Analytic Libraries: A vast majority of all investigated free apps are using advertising and analytic services. While the main purpose of *advertising* services is to monetize apps by displaying targeted ads to its users, *analytic* libraries are meant for providing detailed statistics on how, when and where an app is used. In detail, we observed that 39.9 % of all investigated apps make use of at least one tracking service. Among these, the services provided by Flurry (21.8 %), Google Analytics (10.2 %) and Google AdMob (7.5 %) where the most prevalent (see Table I). As almost every second app uses tracking and advertising services provided by either Flurry or Google, these tracking providers are able to track a user's behavior and activities even across apps. This is of particular interest from a privacy perspective, as providers like Google have the ability to establish a link between a device's hardware identifier and the real identity of its owner. Once authenticated within any Google service on an iOS device, Google would be capable of tracking a user's activities from within all apps, in which their libraries are integrated.

TABLE I: Amount of tracking and advertising libraries used within the 1,336 top free apps (reduced to the top 10 most prevalent libraries).

Library	**# Apps**
Flurry	248 (21.8%)
Google Analytics	116 (10.2 %)
Google AdMob	85 (7.5 %)
Tapjoy	64 (5.6 %)
Chartboost	35 (3.0 %)
Testflight	29 (2.6 %)
Localytics	20 (1.8 %)
Mixpanel	10 (0.9 %)
MobileAppTracker	9 (0.8 %)
Apsalar	8 (0.7 %)

TABLE II: Top 10 of the most frequently requested domains ordered by the number of apps.

Domain	**# Apps**	**# Requests**
`flurry.com`	257 (23.15 %)	588 (5,41 %)
`facebook.com`	155 (13.96 %)	377 (3,47 %)
`admob.com`	127 (11.44 %)	157 (1,44 %)
`apple.com`	75 (6.76 %)	164 (1,51 %)
`tapjoyads.com`	65 (5.86 %)	216 (1,99 %)
`crashlytics.com`	63 (5.68 %)	63 (0,58 %)
`ioam.de`	62 (5.59 %)	174 (1,60 %)
`amazonaws.com`	40 (3.60 %)	141 (1,30 %)
`chartboost.com`	37 (3.33 %)	91 (0,84 %)
`googleadservices.com`	37 (3.33 %)	148 (1,36 %)

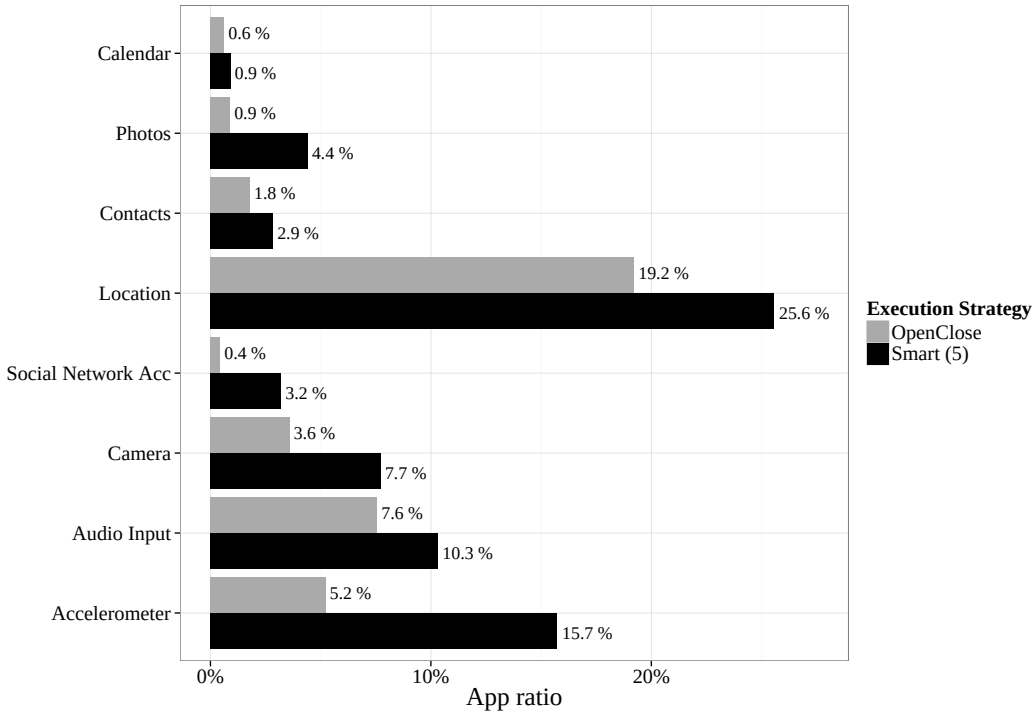


Fig. 4: Amount of apps accessing sensitive API during startup compared to five minutes of Smart Execution.

3) Network Communication: We found 77.7 % of all investigated apps communicating via HTTP within the first 30 seconds after startup. Almost two third (71.2 %) of all communication is unencrypted. Furthermore, we noticed that a large part of all mobile communication is attributed to advertising networks. In detail, almost one fifth (18.7 %) of all requests are addressed to only a limited number of domains, mostly servers of advertising services, as presented in Table II. Moreover, it is worth mentioning that almost 14 % of all apps under review are transmitting data to Facebook's servers. Again, these figures confirm the dominance and omnipresence of mobile advertising.

D. Discussion

As our results demonstrate, mobile advertising has become well-established in recent times. While mobile devices provide a suitable platform for target-oriented advertising, least because of their tight integration in our daily lives, the privacy concerns are obvious. This is why Apple introduced new privacy controls with the release of iOS 7 such as eliminating access to several hardware identifiers like the *Unique Device ID* (UDID) or the Wi-Fi MAC address. Though given the long history of tracking identifiers in iOS, it remains to be seen if deprecating the UDID and eliminating the Wi-Fi MAC address is sufficient to push developers to use Apple's preferred software-based advertising identifiers. We presume no significant changes in a short-term, as advertisers had enough time to correlate a user's hardware identifier with self-defined tracking cookies. With regard to other privacy aspects of iOS, we observe a self-contradictory development. While apps are supposed to ask a user for permission to access the microphone since iOS 7, the camera is still accessible from any app without any restrictions. This still allows any app to take pictures and to transfer them off the device without users consent and knowledge.

For the moment, the main goal of our study was to demonstrate the practicability of DiOS. Therefore, we focused on the amount of sensitive API methods accessed upon launch and compared the results to a five minute execution run of each app. We noticed that most of the privacy-related APIs are called within the first seconds after an app is launched. We assume that even apps with malicious intents behave in the same way, as from a psychological perspective it is reasonable that users might be more willing to grant permission straight after the initial startup, being curious about whether an app delivers what it promises. However, future studies must provide evidence on this assumption.

V. Summary and Future Work

With the exploding number of third-party apps available in the Apple App Store the risk of malicious apps accessing personal information rises. This development demands for automated analysis solutions in order to facilitate empirical studies on the security and privacy landscape of iOS mobile apps.

To address this need, we presented *DiOS*, a fully-automated dynamic analysis solution for large-scale analysis of iOS apps. DiOS features a scalable framework, to schedule apps from the App Store to any number of connected iOS devices. We proposed a novel approach to exercise an app's user interface and to simulate user interaction as realistic as possible. We evaluated different execution strategies provided by DiOS and showed that the coverage rate of our automatic UI exploration approaches is comparably high as within common manual app usage.

To demonstrate the practicability and reliability of DiOS, we utilized its plugin architecture and analyzed 1,136 of the most popular free apps from the App Store. We tracked access to sensitive API methods and monitored related network activities. We found out that almost 80% of all observed apps send and receive data within the first seconds after launch (two third of it unencrypted). Almost half of all apps are sharing data with statistic and tracking libraries and therefore access hardware identifiers in order to reliably track users. As only few tracking providers are prevalent in a high percentage of apps, user's might be tracked across apps. These results are consistent with past studies [16].

In addition, our results indicate a significant privacy problem within the current iOS permission model. Many apps take advantage of the fact that a user's permission to access selected personal data is required only once. This effect can be observed especially within access to location services: One in every five apps accesses the current location on every app launch. Once granted an app like, e.g., Facebook permission to intentionally access location services (e.g. in order to display nearby restaurants) it will track a user's current position on every future app start without any further notifications.

Future Work: One shortcoming of DiOS' *automation* capabilities is the missing handling of registration and authentication forms. Therefore, future versions of DiOS might provide features to fill in forms in a context-sensitive way and to request manual assistance when the method coverage of an app under review does not increase although different execution strategies have been applied.

A major drawback of the DiOS *analyzer* is its missing ability to determine if accesses to privacy-related API really constitute a privacy leak. Implementing taint tracking on the closed-source iOS (where apps are executed natively compared to other platforms) would require either a full system emulation [30] or binary instrumentation [31], [32]. In general, taint tracking as well as detecting malware or vulnerabilities in native code that are only exploited in certain trigger conditions was far beyond the scope of the DiOS analysis platform. In our opinion this remains a research topic even apart from mobile malware. In fact, DiOS opens the doors for such new iOS based research questions and provides a solid basis for future work (not least because of its pluggable architecture).

Therefore, apart from taint tracking, a next step would be to extend the DiOS analyzer to detect more subtle techniques of malicious apps. Recently, Han et al. [18] demonstrated a prototype malware app for iOS that was mainly based on dynamically loading private frameworks at runtime. This is where dynamic analysis techniques, like those provided by DiOS, are more suitable than static analysis approaches [16].

One general limitation of all existing approaches to perform large-scale analysis of iOS apps from the official App Store is its *jailbreak* dependency. Regardless if an app is analyzed statically [16] or dynamically, a jailbreak is required to defeat Apple's FairPlay digital right management technology in order to decrypt an app's code at runtime. Although, most of the DiOS client components currently require a jailbreak, the majority of our techniques could be re-used on non-jailbroken devices, as our automation is mainly based on standard tools provided by Apple. Solely the analyzing features of DiOS would have to be revised in order to place hooks not at runtime, but by modifying an app's binary.

In summary, we believe that apart from tracking down rogue apps and malware, reviewing well-known and popular apps for permission misusage is at least as important. However, along both avenues, much remains to be explored.

References

[1] A. Thampi. Path uploads your entire iPhone address book to its servers. http://mclov.in/2012/02/08/path-uploads-your-entire-address-book-to-their-servers.html.

[2] J. Leyden. D'OH! Use Tumblr on iPhone or iPad, give your password to the WORLD. http://www.theregister.co.uk/2013/07/17/tumblr_ios_uncryption/.

[3] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

[4] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *Security & Privacy, IEEE*, vol. 5, no. 2, pp. 32–39, 2007.

[5] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 116–127.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." in *OSDI*, vol. 10, 2010, pp. 255–270.

[7] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: automatic security analysis of smartphone applications," in *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 2013, pp. 209–220.

[8] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: Having a deeper look into Android applications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 1808–1815.

[9] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 93–104.

[10] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung, "Vision: automated security validation of mobile apps at app markets," in *Proceedings of the second international workshop on Mobile cloud computing and services*. ACM, 2011, pp. 21–26.

[11] P. Gilbert, B.-G. Chun, L. Cox, and J. Jung, "Automating privacy testing of smartphone applications," Technical Report CS-2011-02, Duke University, Tech. Rep., 2011.

[12] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[13] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[14] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna, "Challenges for dynamic analysis of iOS applications," in *Open Problems in Network Security*. Springer, 2012, pp. 65–77.

[15] M. E. Joorabchi and A. Mesbah, "Reverse engineering iOS mobile applications," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 177–186.

[16] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications." in *NDSS*, 2011.

[17] Apple Inc. UI Automation JavaScript Reference. http://developer.apple.com/library/ios/#documentation/DeveloperTools/Reference/UIAutomationRef/.

[18] 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*. Springer, 2013, pp. 272–289.

[19] Apple Inc. iOS Security. [Online]. Available: http://images.apple.com/iphone/business/docs/iOS_Security_Oct12.pdf

[20] J. Freeman. Mobile Substrate. http://www.cydiasubstrate.com.

[21] Apple Inc. Instruments User Guide: Automating UI Testing. [Online]. Available: http://developer.apple.com/library/ios/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UsingtheAutomationInstrument/UsingtheAutomationInstrument.html

[22] ——. iOS Human Interface Guidelines. [Online]. Available: http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/mobilehig/Introduction/Introduction.html

[23] ——. iTunes RSS Feed Generator. https://rss.itunes.apple.com.

[24] libimobiledevice. http://www.libimobiledevice.org.

[25] Apple Inc. NSDistributedNotificationCenter. [Online]. Available: https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSDistributedNotificationCenter_Class/Reference/Reference.html

[26] 148Apps.biz: App Store Metrics: Count of Application Submissions. http://148apps.biz/app-store-metrics/?mpage=submission.

[27] V. van der Veen, H. Bos, and C. Rossow, "Dynamic Analysis of Android Malware," 2013.

[28] J. Krumm, "A survey of computational location privacy," *Personal and Ubiquitous Computing*, vol. 13, no. 6, pp. 391–399, 2009.

[29] P. Marquardt, A. Verma, H. Carter, and P. Traynor, "(sp)iPhone: decoding vibrations from nearby keyboards using mobile phone accelerometers," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 551–562.

[30] E. Bosman, A. Slowinska, and H. Bos, "Minemu: the world's fastest taint tracker," in *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, ser. RAID'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23644-0_1

[31] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the ARM architecture," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 261–270. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176793

[32] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO '08. New York, NY, USA: ACM, 2008, pp. 74–83. [Online]. Available: http://doi.acm.org/10.1145/1356058.1356069