False Sense of Security: A Study on the Effectivity of Jailbreak Detection in Banking Apps

Ansgar Kellner, Micha Horlboge, Konrad Rieck, and Christian Wressnegger
Institute of System Security, TU Braunschweig

***Abstract*—People increasingly rely on mobile devices for banking transactions or two-factor authentication (2FA) and thus trust in the security provided by the underlying operating system. Simultaneously, jailbreaks gain tremendous popularity among regular users for customizing their devices. In this paper, we show that both do not go well together: Jailbreaks remove vital security mechanisms, which are necessary to ensure a trusted environment that allows to protect sensitive data, such as login credentials and transaction numbers (TANs). We find that all but one banking app, available in the iOS App Store, can be fully compromised by trivial means without reverse-engineering, manipulating the app, or other sophisticated attacks. Even worse, 44 % of the banking apps do not even try to detect jailbreaks, revealing the prevalent, errant trust in the operating system's security. This study assesses the current state of security of banking apps and pleads for more advanced defensive measures for protecting user data.**

I. Introduction

An increasing number of people use mobile phones for online banking [1, 2] and as an additional factor of authentication [3]. It is thus of utmost importance that apps offering such services operate in a trusted environment, which ensures the confidentiality and integrity of sensitive data. To this end, vendors heavily rely on security mechanisms provided by the operating system. For instance, at boot time the integrity of all system components is verified and only trusted apps can be executed. Whether a third-party app is trusted or not, in turn, is usually governed by a central authority: In case of iOS and Android, the Apple's App Store and the Google Play Store.

This and various other provisions inevitably constrain the user's ability to customize the device and limit the freedom of choice when installing apps. In consequence, rooting or jailbreaking has become a widely used practice to unshackle a device from the imposed restrictions. According to the developers of Cydia, an alternative to Apple's App Store for jailbroken devices, more than 30 million devices are using their system [4]—hence, are jailbroken. Moreover, the past has shown that shortly after the release of a new iOS version, also a successful and easily accessible jailbreak is not long in coming. For instance, for the most recent version of iOS (12.1), which has been released on October 30, 2018, a jailbreak has been announced [5] and publicly demonstrated on the latest iPhone [6] only a week after.

However, rooting or jailbreaking a device is a privilege escalation attack that also involves the removal of essential security measures of the underlying operating system, and thus, opens the gates for adversaries [7]. While attacks against non-jailbroken devices have been very well feasible in the past [8–10], devices with their primary security mechanisms in place are less likely to be infected by malware or spyware. Such attacks require a high-profile exploit to jailbreak the device in the process, in order to escalate privileges to a comparable level. Fully automated jailbreaks are rare, though. Hence, vendors of third-party apps are particularly interested in detecting jailbreaks to ensure operating in an environment with clearly defined security conditions that allows to keep sensitive data private. This is particularly crucial for banking apps and two-factor authentication.

In this paper, we systematically investigate the effectivity of currently used jailbreak detection mechanisms in banking apps. To this end, we first discuss the different security measures employed by the iOS operating system, types of jailbreaks and jailbreak detection mechanisms in detail. Based on this, we then take a look on means of how to evade jailbreak detection mechanisms and explore how difficult it is to put such attacks into practice. After all, the reliable detection of jailbreaks constitutes the last line of defense of banking apps, ensuring the security of customers by verifying that the key security measures are in place.

Unfortunately, we find that (a) not even all banking apps make use of jailbreak detection, and (b) the large majority of those that do, can be easily evaded. In particular, 15 out of 34 banking applications do not use this vital security measure at all. Apps that attempt to detect a jailbroken device fail in doing so in all but one of the cases and can be evaded by rather simplistic means: We dynamically hook function calls responsible for common detection mechanisms using a widely spread toolkit, Cydia Substrate, and alter the outcome to lead the app to believe in running on a vanilla (no jailbreak applied) iOS. Note, that this functionality comes packaged with Cydia and does neither require any reverse-engineering or manipulations of the app [11], nor sophisticated attacks against TLS pinning [12] or similar. This enables us to record user input from keyboards, such as user credentials (name and password), but also to intercept photos from the built-in camera as used, for instance, for PhotoTANs to verify banking transactions, often used as second factor of authentication.

Additionally, we investigate whether this distressing lack of security (44 % of the banking apps do not even try) is an isolated phenomenon or whether it is representative for the majority of apps in the App Store. We gather a total of 3,482 apps from the official app store across all app genres and widen the testing methodology: First, we look for static strings, such as `jailbreak` or `jailbroken` in the binaries

and identify 2,357 apps (68 %) that supposedly make use of some sort of jailbreak detection. We further narrow down our findings by applying the same dynamic analysis as used for the banking apps and identify that 59 % of all apps in our dataset make use of various forms of jailbreak detection mechanisms. The difference between static and dynamic analysis suggests that either large amounts of apps are shipped with inactive jailbreak detection mechanisms or that these apps use more sophisticated detection mechanism than those observed and evaded in the banking apps—particularly the latter is troublesome.

In summary, we make the following contributions:

- **Overview of Jailbreak Detection and Evasion.** We provide a systematic overview of the different security concepts used in iOS and detail how these relate to the use of jailbreaks in practice. Moreover, we inspect different jailbreak detection mechanisms and how these may be evaded.
- **Security Evaluation of Banking Apps on iOS.** We investigate the effectivity of jailbreak detection as employed by major banking apps and reveal a disastrous current state: Either no detection method is implemented at all or implementations are inadequate, allowing to intercept and record sensitive information.
- **Prevalence Analysis of Jailbreak Detection.** We crawl the Apple App Store and collect the 3,482 most popular apps across genres to analyze whether the lack of security mechanisms observed in banking apps is reflected in other domains as well. Surprisingly, the number of banking apps using jailbreak detection matches the overall average.

The remainder of the paper is structured as follows: Section II reviews basic security concepts of iOS, before Sections III and IV detail the different types of jailbreaks as well as methods to detect them. Section V then deals with ways to evade jailbreak detection mechanisms in practice. Our evaluation, based on banking apps collected from the Apple App Store, is presented in Section VII. Finally, we look upon related work in Section VIII. Section IX concludes the paper.

II. iOS Security

To protect iOS against attackers, Apple provides several security mechanisms that aim at preventing unwanted modifications of the operating system or installed apps. Many of these are rooted at the lowest level and are directly enabled when booting the device. In the following, we briefly discuss the most prominent mechanisms that affect jailbreaks [13]:

Secure Boot Chain. iOS establishes a chain-of-trust in order to verify the integrity of the individual components, from booting the device up to the execution of third-party apps. Figure 1 shows the individual components: The *Boot ROM* represents the chain's root and is implemented immutable as part of the processor and thus, cannot be updated or modified. Due to the use of read-only memory, implicit trust can be presumed. Moreover, it contains the *Apple Root Certificate* including the public key that is used to verify the signature of the *iBoot* boot loader. On devices using an A9 or earlier processors, the *Low Level Bootloader (LLB)* is firstly executed which, in turn, checks the signature of *iBoot*, which then verifies the integrity of the *iOS Kernel*. For later models, the LLB is not used so that the Boot ROM will directly check the signature of iBoot within the boot chain. If at any point in the boot process a signature is missing or fails to be verified, the boot process stops and any further execution of program code is prohibited. The device then switches into recovery mode, demanding to reinstall its firmware. If however the integrity is constituted and the kernel has been successfully booted, apps with a valid signature may start.

After completing the boot process there is no further verification of the chain-of-trust. During runtime modifications to the kernel are possible, for instance, in order to deploy a jailbreak (cf. Section III).

Signed Apps. Before any third-party app is published in the official App Store, it is examined by Apple [14] and is required to be signed with the developer's signature to enable the execution under a functional iOS secure boot chain. This one-time examination checks for obvious flaws or bugs as well as compliance to the App Store review guidelines [15]. Any app that does not comply the latter is rejected, and hence will not be listed on the official App Store. In comparison to Android, it is not possible to "sideload apps", that is, installing apps bypassing the official App Store.

Sandbox. Each third-party app is executed in a unique sandbox that is strictly separated from the sandboxes of other apps as well as the operating system itself. For each app, a randomly named directory is created during installation, for which it owns all rights, including modifying and removing files. Access to everything outside an app's sandbox directory is however forbidden.

Special directories (e.g., the photo directory) can solely be accessed via dedicated services. Access is granted by the operating system and thus, can be revoked at any time. The permissions to access iOS services must be requested via so called "entitlements" during the creation of an app [16]. All requested permissions are bound to the app's signature; thus, cannot be changed without invalidating the signature. Moreover, the sandbox prevents the execution of system calls such as `fork` and `kill`.

Restricted Apps. In order to additionally guard apps from manipulations at runtime, iOS allows to mark apps as "restricted" i.e., forbidding the linker to dynamically load libraries at runtime by specifying the `DYLD_INSERT_LIBRARIES` environment variables [17]. To this end, the linker adds a new segment, called `__RESTRICT`, to the binary that, in turn, contains a section named `__restrict`. Moreover, apps that make use of the `setuid` or `setgid` functions are implicitly tagged as "restricted". This restriction of dynamic linking is orthogonal to the secure boot chain and provides a further layer of protection.

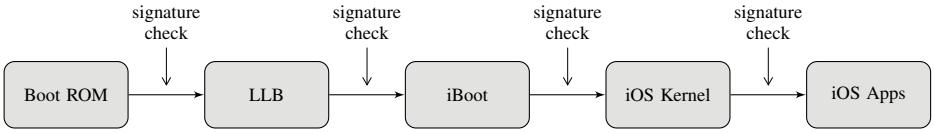

Fig. 1: Schematic depiction of iOS' chain-of-trust.

Further Security Mechanisms. Next to these central security concepts iOS employs mechanisms to impede attacks on the system itself: First, all third-party apps are executed by the non-privileged user `mobile` that has only limited permissions. It is impossible to increase a user's privilege using any of the iOS APIs. Second, the partition that contains the operating system is mounted read-only that means any write access is blocked. Third, *Address Space Layout Randomization (ASLR)* [18] is activated to thwart attacks such as Return Oriented Programming (ROP) [19–21]. Besides, *Pointer Authentication Codes (PAC)* are activated as additional protection against the modification of function pointers and return addresses. Fourth, the *Execute Never (XN)* functionality of ARM processors [22], that implements Data Execution Prevention (DEP), is enabled to avoid the execution of infiltrated code from data memory pages. Fifth, after the initialization of the iOS kernel, the *Kernel Integrity Protection (KIP)* is activated that prevents modifications to the kernel and loaded drivers.

III. Jailbreaks

A jailbreak is a privilege escalation attack that removes the software restrictions of iOS as imposed by Apple. On Android, for instance, similar techniques are known as "rooting the device" [23]. The main goal of a jailbreak is to gain unrestricted access to a device, and thus enable its full customization. This includes the installation of apps from alternative stores, modifications to the user interface and access to the underlying file system. Often jailbreaks are erroneously equated with unlocking the device. While unlocking aims at removing a device's restriction to a particular cellular carrier, jailbreaks lever out the operating system's restrictions with respect to arbitrary software modifications. Both mechanisms, however, are often applied together.

TABLE I: Types of jailbreaks.

	Jailbreak	Dual-boot	Permanent
Tethered	External	–	●
Semi-tethered	External	●	–
Untethered	App	–	●
Semi-untethered	App	●	–

Due to the chain-of-trust that is enforced by iOS, a jailbreak needs to be applied during the boot process, which in turn requires patching the kernel. In practice, there are four types of jailbreaks that differ in whether an external device/computer is needed to boot up and whether the jailbreak persists reboots [24]. Table I provides an overview of the these types.

1) *Tethered Jailbreaks* modify the boot process, but require a computer to enable the jailbreak for each single boot-up. If the device is started without this external factor, the kernel is not patched and the device likely ends up in a partially booted state, such as the iOS recovery-mode.
2) *Semi-tethered Jailbreaks*, in contrast, can be started without applied jailbreak and boot up into stock iOS, without ending up in recovery-mode. However, modified program code and third-party apps cannot be used anymore until the device is restarted again with an external jailbreak tool.
3) *Untethered Jailbreaks* allow to boot a jailbroken device without the help of a computer. During boot up the kernel is automatically exploited such that the device is permanently jailbroken. This type of jailbreak is especially difficult to implement and requires a particular powerful exploit.
4) *Semi-untethered Jailbreaks* are not persistent across reboots, but similar to semi-tethered jailbreaks, the stock iOS functionality remains intact. However, unlike for tethered jailbreaks usually a jailbreak app is installed on the iOS device, which can be launched by the user to patch the kernel again. For iOS 12 beta and iOS 11.4 even a Browser-based exploit is available, allowing the jailbreak to be applied by simply visiting a specially crafted website [25].

Irrespectively of the specific type, most jailbreaks install Cydia, a package manager for alternative apps and extensions that are not available in the official App Store [26]. Cydia essentially is a port of the "Advanced Package Tool" (APT) [27] for iOS devices, bundled with a graphical user interface to assist the installation of apps, and a few tools for manipulating apps.

Cydia Substrate [28], for instance, enables third-party developers to patch existing apps at runtime. It consists of three main components: the MobileHooker, the MobileLoader and the safe mode. The *MobileHooker* enables the hooking of arbitrary system functions that is replacing a function's original implementation with custom program code that may include user-controlled functionality as well as the execution of the original program code. This allows to log API calls, alter return values or any other sort of modification. The

MobileLoader, on the other hand, loads and applies third-party patches for apps at runtime. To this end, it injects itself using the `DYLD_INSERT_LIBRARIES` environment variable, and subsequently loads all other extensions from the dynamic library path. Furthermore, the MobileLoader installs a safety net to pass control over to *safe mode*, whenever an extension crashes the iOS home screen. In safe mode all third-party extensions are disabled and the home screen is restarted.

IV. Jailbreak Detection

Banking apps and other mobile applications that process sensitive user data are well-advised to apply jailbreak detection to ensure they operate in a secure and trusted execution environment. While users benefit from jailbreaks to a certain extend, Apple—with good reason—strongly discourages its use due to security concerns and the fact that the integrity of sensitive data cannot be guaranteed anymore. With jailbreak detection in place, an app may refuse to work and quit if basic security guarantees of the operating systems are not met. In the following, we discuss a few simplistic mechanisms that are frequently used for the detection of jailbreaks in recent iOS apps.

Foreign Files/Apps Checks. One of the simplest ways to detect a jailbreak is to check for apps, tools or files that are commonly installed on jailbroken devices, but that are not present on the default iOS installation. Pre-installed iOS apps are located in the `/Applications` directory, while apps from the App Store are installed to individual, sandboxed directories. Consequently, whenever additional apps are found in the `/Applications` directory this is a strong indicator for a jailbroken device. However, since the set of pre-installed apps differs from version to version, detecting deviations requires separate content lists for each iOS version. Therefore, detection mechanisms usually simply check for the existence of certain apps, for instance `/Applications/Cydia.app`, known to be part of most jailbreak installations. Apart from additional apps, Cydia stores several files on the device that are not present on a vanilla iOS installation. This includes binaries of console tools as well as their corresponding configuration and log files. For example, Cydia Substrate makes use of the dynamic library `MobileSubstrate.dylib` which is located in `/Library/MobileSubstrate/` and that is available on all Cydia installations. From the existence of this library one can thus conclude that (a) Cydia has been installed, and (b) write permissions have been granted outside the sandbox, both only possible on jailbroken devices. Moreover, `_dyld_get_image_name` can be used to list all dynamically linked libraries including those from MobileSubstrate. The situation is similar for certain system services, for instance, the ssh daemon `sshd` which is not installed on vanilla iOS.

File System Checks. iOS maintains two disk partitions: one system partition for the operating system and its components, and a second partition for user-installed apps and data (see Figure 2). While the former is mounted read-only and comparatively small, the latter uses the remaining space of the device and allows arbitrary file access within the limits of each sandbox. To circumvent the restriction of the system partition, directories containing data of interest (wallpapers, apps, etc.) are often moved to the data partition and referenced via symbolic links. The use of symbolic links for system directories are thus a hint for a jailbroken device.

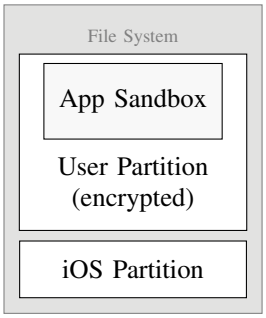

Fig. 2: Schematic depiction of the iOS file system.

As described in Section II, the sandbox of each app forbids write access to any files outside the sandbox—in particular, only the document directory of the app is writable. As result, if an app is able to write outside its sandboxed environment, a jailbreak must have been applied. A common detection mechanism hence involves an attempt to write to the `/private` directory using miscellaneous filenames.

Platform Functionality Checks. For the communication between apps iOS registers individual URL schemes. By invoking the `openURL` function with an app-specific URL, a callback function within the target app is triggered to handle the request. Cydia also registers such a scheme (`cydia://`) to provide a list of installed packages and applied tweaks. The availability of this scheme can simply be checked using the `canOpenURL`, in case of success the device is jailbroken. Besides, the use of system functions is strictly limited on iOS, with some being forbidden at all. For example, an unpatched iOS always returns -1 on invoking `fork`, indicating a failed attempt to create a child process. Some jailbreaks bypass this restriction, such that `fork` successfully spawns a new child process and returns a non-negative process ID. Another system function that can be used as jailbreak indicator is the `system` function: Providing `NULL` as argument can be used to check for the existence of `/bin/sh`. However, an unmodified iOS always returns 0 as the access to the `system` function is restricted, while on a jailbroken device the call correctly resolves to `/bin/sh` and hence returns 1.

V. Evading Jailbreak Detection

In the following, we investigate means of evading jailbreak detection mechanisms that may be used to make iOS apps believe they run in an unmodified, secure environment though they are not. The evasion strategies discussed herein serve as basis for our analysis of the security of banking apps in Section VI.

For evading jailbreak detection there essentially exist three strategies: (a) adapt the jailbreak so that the detection mechanism is not applicable anymore, (b) alter the app that tries to detect the jailbreak, for instance to remove runtime checks, or (c) at runtime, trick the detection mechanism in believing it is running on an unmodified iOS.

Adapting jailbreaks is possible, but likely only effective up to the next app release—resulting in a game of cat-and-mouse between both sides. Modifying program code is error-prone, effortful and specific to individual apps. Thus, the alternative with the widest outreach is the modification at runtime, using tools such as the tweaks of Cydia Substrate [28]—the approach that we will pursue later on to circumvent the jailbreak detection mechanisms of banking apps.

The tweaks allow to hook the underlying system functions and alter return values such that the device appears non-jailbroken. While the basic functionality of hooks is processed by the Objective-C Runtime, different frameworks offer a simplified usage. *Logos* [29], for instance, abstracts the corresponding calls with the commands `%hook`, `%hookf` and `%orig`.

In the following, we will discuss four different hook-based attack examples, each targeting one of the different jailbreak detection mechanism categories from Section IV:

For Apple's Foundation framework [30] three classes are of particular interest for evading jailbreak detection: the `NSFileManager`, the `NSString` and the `UIApplication` class. In the `NSFileManager` class the `fileExistAtPath` function should return `false` for checks on files used in a jailbreak, for all other files the result remains unchanged.

```
%hook NSFileManager
-(BOOL)fileExistsAtPath:(NSString*)path {
   if ([path isEqualToString:@"/Applications/
     Cydia.app"]){
       return NO;
   }
   return %orig;
}
%end
```

Fig. 3: Hooking the `NSFileManager` class.

Figure 3 shows how this is done in practice: The definition of the hook, that is, everything between the `%hook` and the `%end` keywords, specifies the class to address (`NSFileManager`), the function that should be hooked (`fileExistsAtPath`), prefixed with '–', and the Objective-C program code that should be executed. If the Cydia app exists the function always returns `NO`, and the original value denoted with `%orig` otherwise. This equally works for class member fields, such as `attributesOfItemAtPath` that is often used to check for symbolic links and thus always needs to contain the `NSFileTypeRegular` attribute for system folders. Moreover, the modification dates must be set back in time. The `writeToFile` and `writeToURL` functions of `NSString` get modified such that write attempts outside the sandbox are blocked for apps known to make use of jailbreak detection. In `UIApplication` the `canOpenURL` function in turn should refuse requests from apps that are using the `cydia://` scheme by returning `false`. Apart from these functions, some of the system functions need to be addressed as well to cover the remaining detection schemes discussed in Section IV. This includes the `lstat` function to check for symbolic links and the `fopen` function to check for the existence of files.

Figure 4 shows a practical example of how to hook the `lstat` function. In comparison to complete classes, individual functions are handled by the `%hookf` macro function, where the first argument denotes the original function's return type (`int`), the second the name of the function to hook (`lstat`), followed by the types of the remaining function arguments. The implementation identifies all paths that should not be symbolic links on an unpatched iOS device and returns the original field `%orig` with `st_mode` set to `S_IFREG`, denoting a regular file. Otherwise, the original value is returned as it is. Here, the dictionary `shouldNotBeSymbolicLink`, contains all files to be checked.

```
%hookf(int, lstat, const char* path, struct
     stat* buffer){
   NSString* s = [NSString stringWithCString:
     path
              encoding:[NSString
     defaultCStringEncoding]];
   if ([shouldNotBeSymbolicLink containsObject:
     s]) {
      int newMode = (buffer->st_mode ^ S_IFLNK)
     |S_IFDIR;
      buffer->st_mode = newMode;
      return 0;
   }
   return %orig;
}
```

Fig. 4: Hooking the `lstat` function.

The procedure for other system functions is equivalent. Any call to the `system` function with `NULL` as parameter should return -1, since `/bin/sh` is only available on jailbroken devices (see Figure 5).

```
%hookf(int, system, const char* command) {
    if (command == NULL) {
        return 0;
    }
    // ...
  return %orig;
}
```

Fig. 5: Hooking the `system` function.

Additionally, the `fork` system function must be hooked: instead of returning the process ID of the new child process, always -1 is returned, indicating that no child process has been created, as depicted in Figure 6.

Furthermore, the `system` function that can be used to check the availability of a valid shell by specifying `NULL` as command argument must be hooked. The `system` function returns a nonzero value if a shell is available, or 0 otherwise. Consequently, we always return 0 in order to thwart detection.

```
%hookf(pid_t, fork) {
    return -1;
}
```

Fig. 6: Hooking the `fork` function.

Finally, the results of `_dyld_get_image_name` needs to be filtered such that no jailbreak-related dynamic libraries are listed.

All these modifications, that are realized as hooks, require to dynamically load libraries using the `DYLD_INSERT_LIBRARIES` environment variable. If however the binary is marked as "restricted" (cf. Section II) this is not possible. Thus, to equally address all apps that employ jailbreak detection, the signature of the apps must be stripped, the string `__restrict` must be replaced, and the app must be signed again. In contrast to patching specific functions of individual apps, this can be done automatically and for all apps likewise [31].

VI. The Security of Banking Apps

Banking apps provide a convenient way for users to access and manage bank accounts online. The wide use of such apps, however, also attracts miscreants that seek easy financial profit, for instance, by redirecting money transfers. Vendors of banking apps are well aware of the sensitivity of the data they process and often attempt to counteract manipulations by introducing jailbreak detection mechanisms. Once a device is jailbroken anybody can rather easily access and modify arbitrary data.

For our case study, we have gathered 34 banking apps from the Apple App Store. To this end, we have crawled the top-lists of the 200 most popular iOS finance apps in Germany as published by Apple [32] and select all apps that enable a customer to interact with a bank or a banking account. A detailed overview of all analyzed banking apps, including the authors and versions, is given in Table VIII. We start our analysis of the banking apps by investigating how sensitive user input can be intercepted and how detection mechanisms can be evaded.

A. Interception of User Input

A banking app is considered insecure and customers should refrain from using it, whenever an attacker succeeds in intercepting user input, such as login credentials or transaction authentication numbers (TANs), often used as second factor of authentication [11, 33]. We begin by categorizing the banking apps in our dataset based on the type of user input that is used for interaction: (1) System keyboard events, used for entering user credentials, (2) touch events, that are necessary to implement custom keyboards and (3) camera events, as used to scan a security QR-code or photoTAN. All these user inputs can be intercepted by an attacker to obtain sensitive information.

We implement Cydia Substrate hooks for each of these categories to record any sort of user input*. Subsequently, we manually examine each app by running them in the manipulated environment to verify that the implemented hooks work as expected and whether any additional countermeasures are in place, such as jailbreak detection mechanisms, that prevent us from intercepting the data. Table II summarizes our findings: from 34 of the examined banking apps, 94 % (32 instances) require text input, such as entering user credentials or TANs, via the system keyboard. The app `com.db.pbc.mibanco` requires user input via a customized keyboard, while 4 of the banking apps make use of the camera, for instance, to scan PhotoTANs. For these apps the camera is always an additional input method to the system keyboard and never used as sole input method.

The `de.ingdiba.ingdibaibanking` app aside, which has been discontinued and hence is not considered any further in our analysis, we have been able to intercept the user input of 15 banking apps from all input categories using Cydia Substrate. These apps do not make use of any form of jailbreak detection that would prevent these hooks, making them trivially attackable. The remaining 18 banking apps show signs of active jailbreak detection mechanisms, thwarting the interception of user input.

B. Evasion of Jailbreak Detection

For the remaining 18 banking apps with jailbreak detection in place, we implemented Cydia Substrate hooks that manipulate the return values of the functions popularly used for detection mechanisms, as discussed in Section V.

To examine the effectiveness of our evasion approach, we manually inspected all evaded apps again to see if the jailbreak detection methods were successfully disabled. Surprisingly, the user input of all but one of the banking apps have been successfully intercepted despite the deployed countermeasures. Solely the `de.co.barclays.barclaycardgermany` app could not be fully evaded, although we have recorded the activation of several of our hooks, including the interception of keyboard events. The app implements one additional and slightly more sophisticated check that detects the indirect jumps (*trampolines*) that are inserted by Cydia Substrate at each hooked function [34]. Additionally, the check takes precautions to skip all `NOP`s, in case an attacker bluntly overwrites these commands. With some little extra effort this check can however be easily bypassed as well.

Our experiments show that with these simple evasion hooks we are able to evade the jailbreak detection mechanisms of 17 out of 18 banking apps. In consequence, we are able to intercept sensitive user information from 94 % of the banking apps (32 of 34 instances). This result shows that a surprisingly high number of banking apps can be tricked into running on jailbroken devices, despite the vendor's countermeasures with rather simplistic means. In this way, we have been able to

*More details about the applied hooks and the corresponding source code can be found on our webpage at: `https://dev.sec.tu-bs.de/ios/`

TABLE II: Overview of the evasion of jailbreak detection mechanisms in all banking apps.

Banking App	JB Detection	Evaded	Interception of		
			Keyboard	Touch Events	Camera
1822direkt	●	✓	●	–	–
AMEX BD	–	–	●	–	–
Audi Banking	●	✓	●	–	–
BBBank-Banking	●	✓	●	–	–
Banking	●	✓	●	–	–
Barclaycard App	●	✗	●	–	–
Consorsbank	–	–	●	–	–
DKB-Banking	–	–	●	–	–
DKB-Card-Secure	●	✓	●	–	●
Degussa Bank Banking + Brokerage	–	–	●	–	–
Deutsche Bank Mobile	●	✓	●	–	–
GLS mBank	●	✓	●	–	–
ING-DiBa Austria Banking App	–	–	●	–	–
ING-DiBa Banking + Brokerage	✗*	–	–	–	–
ING-DiBa Banking to go	–	–	●	–	–
Mi Banco db	●	✓	–	●	–
MoneYou Spar-App HD	●	✓	●	–	–
MyBankingApp	●	✓	●	–	–
OLB photoTAN	●	✓	●	–	●
Online-Filiale+	–	–	●	–	–
Ophirum Gold	–	–	●	–	–
Outbank: Intelligent Banking	●	✓	●	–	–
S-ID-Check	●	✓	●	–	●
Santander MobileBanking	–	–	●	–	–
SpardaSecureApp	–	–	●	–	–
TARGOBANK Mobile Banking	●	✓	●	–	–
Wavy App	–	–	●	–	–
WorldRemit Money Transfer	–	–	●	–	–
Wüstenrot Banking	●	✓	●	–	–
apoBank+	–	–	●	–	–
comdirect banking App	–	–	●	–	–
flateXSecure	●	✓	●	–	●
norisbank mobile	●	✓	●	–	–
vaamo – Die digitale Vermögensverwaltung	–	–	●	–	–

*The development of the app has been discontinued.

intercept user data for the large majority of banking apps. Given the sensitivity of banking and payment information this is particularly troublesome.

VII. Jailbreak Detection Across App Genres

To get a better feeling for the numbers determined in the previous section, we proceed to inspect the prevalence of jailbreak detection mechanisms across application genres and examine whether the use of jailbreaking detection in banking apps is below or above average.

The dataset used for the evaluation is detailed in Section VII-A, before we conduct a number of quantitative measurements, based on static and dynamic analysis: First, we identify all apps that contain suspicious strings that indicate jailbreak detection mechanisms (Section VII-B). Second, we complement our static analysis with dynamic function hooking to provide a detailed view on used mechanisms and their propagation (Section VII-C).

A. The Dataset

For our analysis we have gathered a total of 3,482 apps from the Apple App Store. To this end, we have crawled the top-lists of the most popular iOS apps in Germany, as published by Apple [32]. Table III summarizes our dataset. In total we have collected data from 23 different genres. Many apps are however assigned to multiple genres, meaning that these rankings are not disjoint. We thus attribute each app solely to its primary genre. As consequence, the number of apps per genre is highly unbalanced, which however has no impact on the validity of our evaluation. Subsequently, we inspect all apps of our dataset in detail, focusing on the used jailbreak detection mechanisms in these apps.

TABLE III: Overview of all genres.

Genre	# Apps	Genre	# Apps	Genre	# Apps
Book	177	Lifestyle	170	Reference	165
Business	148	Magazines & News	30	Shopping	147
Education	169	Medical	143	Social Networking	158
Entertainment	164	Music	133	Sports	172
Finance	133	Navigation	153	Travel	167
Food & Drink	162	News	187	Utilities	138
Games	175	Photo & Video	149	Weather	143
Health & Fitness	159	Productivity	140		

TABLE IV: Overview of jailbreak detection checks across all genres.

Genre	**Foreign Files/Apps**		**File System**		**Platform Functionality**		**Any**	
Book	60 %	106	5 %	9	3 %	5	60 %	106
Business	41 %	60	5 %	8	5 %	7	41 %	60
Education	53 %	90	4 %	6	0 %	0	53 %	90
Entertainment	64 %	105	5 %	8	2 %	4	64 %	105
Finance	50 %	67	5 %	6	5 %	6	50 %	67
Food & Drink	43 %	69	2 %	3	2 %	4	43 %	69
Games	86 %	151	39 %	69	10 %	17	86 %	151
Health & Fitness	56 %	89	3 %	5	1 %	1	56 %	89
Lifestyle	59 %	100	8 %	13	1 %	2	59 %	100
Magazines & News	10 %	3	0 %	0	0 %	0	10 %	3
Medical	33 %	47	1 %	1	0 %	0	33 %	47
Music	68 %	90	10 %	13	1 %	1	68 %	90
Navigation	50 %	76	3 %	5	3 %	5	50 %	76
News	53 %	100	1 %	1	2 %	3	54 %	101
Photo & Video	67 %	100	8 %	12	3 %	5	67 %	100
Productivity	56 %	78	6 %	8	1 %	2	56 %	78
Reference	63 %	104	7 %	12	1 %	1	63 %	104
Shopping	72 %	106	5 %	8	3 %	4	72 %	106
Social Networking	66 %	104	6 %	10	4 %	6	66 %	104
Sports	61 %	105	3 %	5	1 %	2	61 %	105
Travel	62 %	103	3 %	5	4 %	6	62 %	103
Utilities	63 %	87	5 %	7	2 %	3	63 %	87
Weather	67 %	96	3 %	5	0 %	0	67 %	96
Total	**58 %**	**2036**	**6 %**	**219**	**2 %**	**84**	**59 %**	**2037**

B. Static Analysis

To get a first idea of how many of the crawled apps include indicators of jailbreak detection mechanisms, we statically inspect the app binaries. The analysis can be conducted externally outside the restricted iOS system environment. However, before starting with the analysis, first, the app binaries has to be decrypted [35], since each app is individually encrypted by the App Store for the specific device it was downloaded to. A small fraction of apps (0.5 %) could not successfully be decrypted and thus, these apps neglected from further analysis.

A simple search on the decrypted app binaries for suspicious strings is conducted that may contain clues about jailbreak detection mechanisms. To this end, we extract all strings from the apps' binaries and then look for obvious substrings such as `jailbreak` or `jailbroken` as well as previously identified foreign apps or files that are usually used in jailbreak detection mechanisms. Surprisingly, using this simple heuristic 68 % of the analyzed apps (2,357 instances) showed one or multiple indicators of possible jailbreak detection mechanisms.

By ignoring any sort of dynamic composing of strings and objects as well as ignoring the apps' program flows, which are a little bit tricky to extract from iOS binaries, it is clear that not all possible jailbreak detection mechanisms can be detected with this simple analysis. Particularly, concatenated strings, relative paths or obfuscated strings cannot be reconstructed with the applied static approach, and thus not be detected. As a result, the found string matches can only be seen as first indicators of possible jailbreak detection mechanisms.

C. Dynamic Analysis

Statically analyzing program code has a number of limitations in practice [36, 37] that render the previous analysis potentially incomplete. Hence, we employ a complementary dynamic analysis in order to increase the coverage of jailbreak detection mechanisms that can be identified. To this end,

we consider the mechanisms described in Section IV and analyze these using dynamic function hooking. We start with an overview of the prevalence of jailbreak detection in our dataset and subsequently highlight details on individual mechanisms.

The Big Picture. For this experiment we consider the three categories of jailbreak detection, as presented in Section IV, and look for their use in our dataset: (1) The availability of foreign files/apps checks, (2) the presence of file system checks, and (3) the use of platform functionality checks. Table IV summarizes the results. The first column states the genre of the app, while the following three columns are associated with the introduced categories of jailbreak detection mechanisms. For each category its relative as well as absolute occurrence is stated. The last column indicates if an app contains at least one jailbreak detection mechanism.

Interestingly, more than half of the examined apps (2,037 instances) make use of at least one jailbreak detection mechanism of any kind. The most popular mechanism for jailbreak detection is the check for foreign apps/files (58 %), actually used in all investigated banking apps. Followed by checks regarding the filesystem (6 %) and checks for platform functionality (2 %). As shown in Table IV some of the apps make use of multiple mechanisms to increase the chance of detecting a jailbreak.

Details on Used Mechanisms. Next, we look at the parametrization of the detection mechanisms to determine what aspects of a jailbroken device apps attempt to detect most frequently.

TABLE V: Most common foreign files/apps checks.

Value	%	Total
`/Applications/Cydia.app`	15 %	2224
`/bin/bash`	13 %	1857
`/Library/MobileSubstrate`	8 %	1151
`/bin/sh`	7 %	1095
`/Library/Frameworks/CydiaSubstrate.framework`	7 %	1028

Table V shows the most frequently checked foreign files/apps. About 15 % of the analyzed apps look for `Cydia.app`, which is bundled with many jailbreak tools. Followed by the two shells, the Bash shell (`bash`) with 13 % and the unix shell (`sh`) with 7 %, and two files of Cydia components the `MobileSubstrate` with 8 % and `CydiaSubstrate` with 7 %.

TABLE VI: Most common file system checks.

Value	%	Total
`/private/jailbreak.txt`	27 %	91
`/private/jailbreak.test`	24 %	81
`/Applications`	15 %	49
`/usr/libexec`	6 %	21
`/usr/share`	5 %	18

The most used file system checks are summarized in Table VI. More than half of the conducted file system checks look for extended write permissions by testing for one of the two files `/private/jailbreak.txt` and `/private/jailbreak.test`. The choice of names is not noteworthy per se, but the fact that these filename are suggested in a number of StackOverflow postings [e.g., 38]: Many developers seem to verbatim 'copy & paste' solutions found online. This also lines up with recent research on security-related code-clones [39]. The other filenames only appear once and thus, seem to be application specific or randomly generated. Also the application directory `/Applications` is often checked (15 %). This make sense because in general this memory intense directory does not fit on the iOS system partition and, thus, must be moved to the data partition and made available via a symbolic link on jailbroken devices. Less checks are performed on `/usr/libexec` and `/usr/share`.

TABLE VII: Most common platform functionality checks.

Value	%	Total
`system`	63 %	64
`cydia://`	15 %	15
`fork`	15 %	15
`cydia://package/com.example.package`	7 %	7
`cydia://package`	1 %	1

In general, only 2 % of the analyzed apps make use of iOS functions to detect jailbreaks (see Table VII). However, when this sort of detection mechanism is applied in more than half of the cases the `system` function is used. The `fork` function is only used in 15 % of the checks. The other checks from this category target the `cydia://` URL scheme in different variants. Due to their similar structure they are likely to be originated from code examples. This internal URL scheme is only available when Cydia is installed.

VIII. Related Work

In recent years, the research community has looked upon various security-related aspects of mobile operating systems. To this end, many have focused on Android systems for analyzing apps [40–43], detecting malware and attacks [44–47], or finding and describing vulnerabilities [48–50]—among many other topics. Moreover, jailbreaking or rooting Android devices is discussed both from an offensive [23, 51, 52] as well as a defensive point of view [11, 33, 53–56]. In this section, we however focus on the iOS platform for which we first discuss attacks on unmodified as well as jailbroken systems. Second, we review defensive measures against adversaries beyond those that are already implemented in the operating system.

Attacks. There exists a plethora of practical work leading to jailbreaks [e.g., 57–59] and non-academic research that discusses exploits and internals of the iOS operating systems [60–64]. Academically, Wang et al. [65] have, for instance, presented an attack scheme (Jekyll) that is able to circumvent Apple's vetting system by triggering the malicious payload remotely, once an app has been successfully added to the App Store. The

app then rearranges existing, signed program code to enable the intended malicious control flow. Since the newly arranged code has not existed during the review process, the vetting system has no chance of detecting malicious payload of that kind. Wang et al. [8], on the other hand, demonstrate the feasibility of infecting a large number of iOS devices through botnets. By exploiting fundamental design flaws in the iTunes syncing system, the device provisioning process and the file storage, an malicious app can be installed that in turn steels private data.

Xing et al. [66] introduce "cross-app resource access" (XARA) attacks, that enable an adversary to obtain access to resources of other apps. Normally, such attempts are rendered impossible by the application sandbox; however, due to the lack of app-to-app and app-to-system authentication for resource interactions this becomes possible. SandScout [67] is a framework to extract, analyze, and formally model sandbox profiles as logic-based programs. The authors introduce Prolog-based queries that are used to evaluate file-based security properties and uncover seven new classes of exploitable vulnerabilities that affect jailbroken and unmodified iOS devices likewise. Apart from that there are a couple of existing Cydia tweaks that try to bypass jailbreak detection mechanisms, among them xCon [68], Liberty [69], JailProtect [70], tsProtector [71] etc. The drawbacks of these solutions are that most of them target specific jailbreak detection mechanisms and iOS versions; additionally, most of the approaches are not available as open source.

Defenses. Privacy is a key concern when it comes to defensive measures of mobile devices. To this end, Egele et al. [31] utilized a combination of control-flow and data-flow analysis to identify privacy leaks in iOS apps. The concept is then extended by Szydlowski et al. [72] to dynamic analysis, in order to mitigate the use of obfuscation techniques in malicious apps. Moreover, they present a tool for the automatic interaction with apps to also analyze functionality that can be triggered through the user interface only. With DiOS, Kurtz et al. [73] present a framework to automatize iOS apps, which allows to simulate user interactions in the context of the started app. While the framework has originally been applied to identify privacy leaks, for our work we make use of the system to automate the collection of iOS apps at a large scale from the official App Store.

To protect against runtime attacks, Pewny and Holz [74] propose the use of control-flow integrity [75]. The system is implemented as an extension to LLVM [76, 77] and thus can be employed for compiling apps targeting various kinds of iOS devices. iRiS [78] is a vetting system for iOS apps implemented on top of Valgrind [79] that uses static analysis to resolve security-critical API calls. In case this fails, the system falls back to iterative dynamic analysis. Chen et al. [80], on the other hand, detect legitimate libraries that were repackaged for propagating malware. In this work, the authors observe that frequently the same libraries are used for Android as well as iOS. Hence, they first analyze the Android version of the libraries to leverage existing tools and only if a malicious library is identified, the corresponding iOS version is investigated, based on invariant features that are shared cross platforms. CRiOS [81] is a fully-automated system to collect vast amounts of iOS applications. On a comprehensive set of iOS apps the authors perform two large-scale analyses: one dealing with the general ratio of third-party libraries in the collected apps and another targeting at the correct use of TLS/SSL certificates in those apps.

Though there has been considerable efforts towards different attacks and defenses for iOS, there has been no academic study on the use of jailbreak detection mechanisms and its evasion yet. To illustrate the practical relevance of the topic, we focus on the evasion of jailbreak detection mechanisms of banking apps, which are a critical target for attackers.

IX. Conclusion

Jailbreaks go along with severe security implications for the execution of third-party apps on iOS. After a jailbreak the operating system cannot guarantee a trusted execution environment anymore and thus, sensitive user data may be at risk. The app vendor's last line of defense hence is the detection of jailbreaks to at least be able to refuse operation under insecure conditions. Due to the predominant use of simplistic detection strategies, this however is frequently not crowned with success.

We show that only 18 out of 34 banking apps from the Apple App Store make use of jailbreak detection mechanisms, while the remaining apps are entirely unguarded against attacks. All but one of those that attempt to detect jailbreaks can be evaded by out of the box function hooking, rendering these mechanisms inadequate for protecting sensitive data. Surprisingly, banking apps do not exceed the average use of jailbreak detection mechanisms across application genres. Given the sensitivity of financial transactions one would expect a significantly higher ratio of banking apps that apply jailbreak detection mechanisms. Moreover, our analysis finds that many application developers apparently copy&paste source code for detecting jailbreaks from community-provided platforms such as StackOverflow, making these detection schemes an easy target. The results of our study urgently call for more advanced jailbreak detection mechanisms to protect against eavesdropping and data theft in two-factor authentication scenarios as used in most banking applications.

Acknowledgment

The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the projects VAMOS (16KIS0534) and FIDI (16KIS0786K).

Availability

To foster future research and improve existing implementations of jailbreak detection mechanisms, we make all developed tools and hooks available at:

`https://dev.sec.tu-bs.de/ios/`

References

[1] B. Reaves, N. Scaife, A. Bates, P. Traynor, and K. R. Butler, "Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world," in *Proc. of the USENIX Security Symposium*, 2015, pp. 17–32.

[2] B. of Governors of the Federal Reserve Division of Consumer and C. Affairs, "Consumers and mobile financial services 2016," 2016.

[3] A. Dmitrienko, C. Liebchen, C. Rossow, and A.-R. Sadeghi, "On the (in)security of mobile two-factor authentication," in *Proc. of the International Conference on Financial Cryptography and Data Security*, 2014.

[4] J. Freeman, "Saurik," http://www.saurik.com/, 2018, visited November, 2018.

[5] C. Liang, "iOS 12.1 + A12 == the end of iOS war?" https://twitter.com/chenliang0817/status/1059871797456396288, 2018.

[6] ——, "Era of iOS 12 with A12: End of iOS war?" Presentation at Power of Community (POC), 2018.

[7] L. García and R. J. Rodríguez, "A peek under the hood of iOS malware," in *Proc. of the International Conference on Availability, Reliability and Security*, 2016, pp. 590–598.

[8] T. Wang, Y. Jang, Y. Chen, S. Chung, B. Lau, and W. Lee, "On the feasibility of large-scale infections of iOS devices," in *Proc. of the USENIX Security Symposium*, 2014, pp. 79–93.

[9] C. Xiao, "AceDeceiver: First iOS trojan exploiting apple DRM design flaws to infect any iOS device," https://researchcenter.paloaltonetworks.com/2016/03/acedeceiver-first-ios-trojan-exploiting-apple-drm-design-flaws-to-infect-any-ios-device/, 2016, visited November, 2018.

[10] Wikileaks, "Vault 7: Cia hacking tools revealed," https://wikileaks.org/ciav7p1/cms/space_2359301.html, 2018, visited November, 2018.

[11] V. Haupert, D. Maier, N. Schneider, J. Kirsch, and T. Müller, "Honey, i shrunk your app security: The state of android app hardening," in *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2018, pp. 69–91.

[12] C. M. Stone, T. Chothia, and F. D. Garcia, "Spinner: Semi-automatic detection of pinning without hostname verification," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2017, pp. 176–188.

[13] Apple Inc., "iOS security – iOS 12," https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf, 2018.

[14] ——, "App review," https://developer.apple.com/app-store/review/, 2018, visited November, 2018.

[15] ——, "App store review guidelines," https://developer.apple.com/app-store/review/guidelines/, 2018, visited November, 2018.

[16] ——, "About entitlements," https://developer.apple.com/library/archive/documentation/Miscellaneous/Reference/EntitlementKeyReference/Chapters/AboutEntitlements.html, 2018, visited November, 2018.

[17] ——, "dyld.cpp," http://www.opensource.apple.com/source/dyld/dyld-210.2.3/src/dyld.cpp, 2004–2010, visited November, 2018.

[18] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *Proc. of the USENIX Security Symposium*, 2003.

[19] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[20] R. Hund, T. Holz, and F. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proc. of the USENIX Security Symposium*, 2009.

[21] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[22] ARM Limited, "Arm v6-m architecture reference manual," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0419e/index.html, 2017.

[23] S. Sun, A. Cuadros, and K. Beznosov, "Android rooting: Methods, detection, and evasion," in *Proc. of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2015, pp. 3–14.

[24] S. et al., "The iPhone wiki," https://www.theiphonewiki.com/wiki/, 2018, visited November, 2018.

[25] H. Linan, L. Liu, and Z. Qixun, "Remote code execution in mobile browser – the mobile Pwn2Own case study," Presentation at Mobile Security Conference (MOSEC), 2018.

[26] J. Freeman, "Cydia," https://cydia.saurik.com, 2018, visited November, 2018.

[27] Debian, "Advanced package tool (apt) - command-line package manager," https://tracker.debian.org/pkg/apt, 2018, visited November, 2018.

[28] SaurikIT, LLC, "Cydia substrate," http://cydiasubstrate.com, 2018, visited November, 2018.

[29] theosdev, "Theos," https://theos.github.io/, 2018, visited November, 2018.

[30] Apple Inc., "Foundation – Apple developer documentation," https://developer.apple.com/documentation/foundation, 2018, visited November, 2018.

[31] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting privacy leaks in iOS applications," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2011, pp. 177–183.

[32] Apple Inc., "iTunes RSS feed generator," https://rss.itunes.apple.com/en-us, 2018, visited November, 2018.

[33] V. Haupert and T. Müller, "On app-based matrix code authentication in online banking," in *Proc. of International Conference on Information Systems Security and Privacy (ICISSP)*, 2018.

[34] N. Labs, "Who owns your runtime?" https://labs.nettitude.com/blog/ios-and-android-runtime-and-anti-debugging-protections/, 2018, visited November, 2018.

[35] KJCracks, "Clutch: Fast iOS executable dumper," https://github.com/KJCracks/Clutch, 2018, visited November, 2018.

[36] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2003, pp. 290–299.

[37] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2007, pp. 421–430.

[38] S. Overflow, "How do i detect that an iOS app is running on a jailbroken phone?" https://stackoverflow.com/questions/413242, 2018, visited November, 2018.

[39] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy&paste on android application security," in *Proc. of the IEEE Symposium on Security and Privacy*, 2017, pp. 121–136.

[40] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 393–407.

[41] B. Bichsel, V. Raychev, P. Tsankov, and M. T. Vechev, "Statistical deobfuscation of android applications," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 343–355.

[42] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic security analysis of smartphone applications," in *Proc. of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[43] M. Spreitzenbarth, F. C. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-Sandbox: Having a deeper look into android applications," in *Proc. of the ACM Symposium on Applied Computing (SAC)*, 2013, pp. 1808–1815.

[44] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Efficient and explainable detection of Android malware in your pocket," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2014.

[45] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "MAMADROID: Detecting android malware by building markov chains of behavioral models," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[46] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli, "Yes, machine learning can be more secure! a case study on android malware detection," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, May 2017.

[47] X. Wang, X. Wang, D. Zhou, and Y. Yang, "Droid-AntiRM: Taming control flow anti-analysis to support automated dynamic analysis of android malware," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2017.

[48] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2013, pp. 73–84.

[49] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: From two permissions to complete control of the ui feedback loop," in *Proc. of the IEEE Symposium on Security and Privacy*, 2017, pp. 1041–1057.

[50] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 356–367.

[51] A. Nazar, M. M. Seeger, and H. Baier, "Rooting android - extending the ADB by an auto-connecting wifi-accessible service," in *Proc of. Information Security Technology for Applications (NordSec)*, 2011, pp. 189–204.

[52] R. Hay, "fastboot oem vuln: Android bootloader vulnerabilities in vendor customizations," in *Proc. of the USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[53] Y. Shen, N. S. Evans, and A. Benameur, "Insights into rooted and non-rooted android mobile devices with behavior analytics," in *Proc. of the ACM Symposium on Applied Computing (SAC)*, 2016, pp. 580–587.

[54] I. Gasparis, Z. Qian, C. Song, and S. V. Krishnamurthy, "Detecting android root exploits by learning from root providers," in *Proc. of the USENIX Security Symposium*, 2017, pp. 1129–1144.

[55] N. S. Evans, A. Benameur, and Y. Shen, "All your root checks are belong to us: The sad state of root detection," in *Proc. of ACM International Symposium on Mobility Management and Wireless Access (MobiWac)*, 2015, pp. 81–88.

[56] J. Jin and W. Zhang, "System log-based android root state detection," in *Proc. of Cloud Computing and Security ICCCS*, 2017, pp. 793–798.

[57] PanGu, "iOS jailbreak tool," http://en.pangu.io/, 2018, visited November, 2018.

[58] O. Community, "Apple tools. built from scratch. for the community." https://github.com/OpenJailbreak, 2018, visited November, 2018.

[59] Dev-Team, "Dev-team blog," http://blog.iphone-dev.org/,

2018, visited November, 2018.

[60] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.-P. Weinmann, *iOS Hacker's Handbook*. Wiley, 2012.

[61] S. Esser, "iOS 10 kernel heap revisited," in *Proc. of Hack in the Box GSEC*, 2016.

[62] N. Dhanjani, "New age application attacks against apple's iOS (and countermeasures)," in *Proc. of Black Hat Europe*, 2011.

[63] T. Mandt, "Revisiting iOS kernel (in)security: Attacking the early random() PRNG," Presentation at CanSecWest, 2014.

[64] geosn0w, "Jailbreaks demystified," https://geosn0w.github.io/Jailbreaks-Demystified/, 2018, visited November, 2018.

[65] T. Wang, K. Lu, L. Lu, S. P. Chung, and W. Lee, "Jekyll on iOS: When benign apps become evil." in *Proc. of the USENIX Security Symposium*, 2013, pp. 559–572.

[66] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han, "Cracking app isolation on apple: Unauthorized cross-app resource access on MAC OS X and iOS," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 31–43.

[67] L. Deshotels, R. Deaconescu, M. Chiroiu, L. Davi, W. Enck, and A.-R. Sadeghi, "SandScout: Automatic detection of flaws in iOS sandbox profiles," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 704–716.

[68] n00neimp0rtant, "xcon," https://github.com/n00neimp0rtant/xCon-Issues, 2018, visited November, 2018.

[69] Y. Jailbreak, "Liberty," https://yalujailbreak.net/liberty/, 2018, visited November, 2018.

[70] J. Verne, "JailProtect," https://julioverne.github.io/description.html?id=com.julioverne.jailprotect, 2018, visited November, 2018.

[71] typ0s2d10, "tsprotector 8+ (ios 8+)," http://moreinfo.thebigboss.org/moreinfo/depiction.php?file=tsprotector8Dp, 2018, visited November, 2018.

[72] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna, "Challenges for dynamic analysis of ios applications," in *Proc. of Open Problems in Network Security – IFIP WG 11.4 International Workshop*, 2011, pp. 65–77.

[73] A. Kurtz, A. Weinlein, C. Settgast, and F. Freiling, "DiOS: Dynamic privacy analysis of iOS applications," Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, Tech. Rep., 2014.

[74] J. Pewny and T. Holz, "Control-flow restrictor: Compiler-based CFI for iOS," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2013, pp. 309–318.

[75] M. Abadi, M. Budiu, and Úlfar Erlingsson, "Control-flow integrity," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2005, pp. 340–353.

[76] LLVM Developer Group, "The LLVM compiler infrastructure," http://llvm.org/, 2018, visited November, 2018.

[77] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of the International Symposium on Code Generation and Optimization (CGO)*, 2004, pp. 75–88.

[78] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iRiS: Vetting private api abuse in iOS applications," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 44–56.

[79] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. of the ACM SIGPLAN International Conference on Programming Languages Design and Implementation (PLDI)*, 2007, pp. 89–100.

[80] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou, "Following devil's footprints: Cross-platform analysis of potentially harmful libraries on Android and iOS," in *Proc. of the IEEE Symposium on Security and Privacy*, 2016, pp. 357–376.

[81] D. Orikogbo, M. Büchler, and M. Egele, "CRiOS: Toward large-scale iOS application analysis," in *Proc. of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2016, pp. 33–42.

TABLE VIII: Overview of analyzed iOS banking apps.

Banking App	Author	Package Name	Version
1822direkt	Innoopract Informationssysteme GmbH	de.direkt1822.banking	2.0.1
AMEX BD	The City Bank Limited	com.thecitybank.myca	1.5
Audi Banking	Volkswagen Financial Services AG	com.vwfs.BankingWebApp.Audi	2.3
BBBank-Banking	Fiducia & GAD IT AG	de.fiducia.iphone.bbb.banking	17.16.02
Banking	Volkswagen Financial Services AG	com.vwfs.BankingWebApp	2.3
Barclaycard App	Barclays Bank Plc	de.co.barclays.barclaycardgermany	1.0.12
Consorsbank	BNP Paribas S.A. Niederlassung Deutschland	de.consorsbank.universalapp	1.10.0
DKB-Banking	Deutsche Kreditbank AG	de.dkb.portalapp	2.4.1
DKB-Card-Secure	Deutsche Kreditbank AG	de.dkb.cardsecure	1.0.0
Degussa Bank Banking + Brokerage	Degussa Bank AG	de.degussa-bank.BankingBrokerage	2.0.0
Deutsche Bank Mobile	Deutsche Bank AG	com.db.pbc.ng.mobile	1.10.0
GLS mBank	GLS Gemeinschaftsbank eG	de.gls.mbank	1.8.7
ING-DiBa Austria Banking App	ING-DiBa Austria	at.ing.diba.client.mobile.ios	3.2.4
ING-DiBa Banking + Brokerage	ING-DiBa AG	de.ingdiba.ingdibaibanking	4.2.8
ING-DiBa Banking to go	ING-DiBa AG	de.ingdiba.bankingapp	1.13.3
Mi Banco db	Deutsche Bank AG	com.db.pbc.mibanco	2.2.16
MoneYou Spar-App HD	MoneYou	nl.moneyou.Spar-AppHD	5.3.1
MyBankingApp	Fiducia & GAD IT AG	de.fiduciagad.iphone.wl.banking	17.16.01
OLB photoTAN	Oldenburgische Landesbank AG	de.olb.phototan	4.13.5
Online-Filiale+	Fiducia & GAD IT AG	de.gad.onlinefilialeplus	3.5.1
Ophirum Gold	Ophirum Commodity GmbH	de.ophirum.app	1.3.0
Outbank: Intelligent Banking	Outbank - The intelligent online banking app GmbH	com.stoegerit.outbank.ios	1.12.2
S-ID-Check	Netcetera AG	com.netcetera.s-id-check	1.1.2
Santander MobileBanking	Santander Consumer Bank AG	mobile.santander.SantanderDE	3.4
SpardaSecureApp	Sparda-Datenverarbeitung eG	de.sdvrz.sparda.secureapp.produktion	2.0.2
TARGOBANK Mobile Banking	Euro-Information	ei.targo.prd	3.29.1
Wavy App	Klarna AB	com.klarna.wavy	1.2.0
WorldRemit Money Transfer	WorldRemit	com.worldremit.ios	3.10.0
Wüstenrot Banking	Wüstenrot & Württembergische AG	de.wwag.wuestenrot.banking.ios	17.16.02
apoBank+	Deutsche Apotheker- und Ärztebank	de.apobank.apobankplus	3.1.1
comdirect banking App	comdirect bank AG	de.comdirect.comdirectibanking	3.5.6
flateXSecure	XCOM AG	de.xcom.PTANFlatex	1.1.0
norisbank mobile	norisbank GmbH	de.norisbank.app.ios.norisbank	3.0.0
vaamo – Die digitale Vermögensverwaltung	Vaamo Finanz AG	de.vaamo.webapp	2.0.1