nccgroup

NCC Group Whitepaper

Faux Disk Encryption: Realities of Secure Storage On Mobile Devices

August 4, 2015 – Version 1.0

Prepared by

Daniel A. Mayer — Principal Security Consultant

Drew Suarez — Senior Security Consultant

Abstract

In this paper, we discuss the challenges mobile app developers face in securing data stored on devices including mobility, accessibility, and usability requirements. Given these challenges, we first debunk common misconceptions about full-disk encryption and show why it is not sufficient for many attack scenarios. We then systematically introduce the more sophisticated secure storage techniques that are available for iOS and Android respectively. For each platform, we discuss in-depth which mechanisms are available, how they technically operate, and whether they fulfill the practical security and usability requirements. We conclude the paper with an analysis of what still can go wrong even when current best-practices are followed and what the security and mobile device community can do to address these shortcomings.

Table of Contents

1 Introduction . . . 3

2 Challenges in Secure Mobile Storage . . . 4

3 Threat Model Considerations . . . 5

4 Secure Data Storage on iOS . . . 6

4.1 Fundamentals of iOS Data Protection . . . 7

4.2 Filesystem Encryption . . . 8

4.3 Keychain . . . 10

4.4 Touch ID . . . 11

4.5 Jailbreaking . . . 12

5 Secure Data Storage on Android . . . 13

5.1 Fundamentals of Android's Secure Data Storage . . . 13

5.2 Full-Disk Encryption . . . 13

5.3 Android Credential Storage . . . 15

5.4 Platform Diversity Considerations . . . 16

5.5 Consequences of Inconsistent Bootloader Security . . . 16

6 Conclusions . . . 19

1 Introduction

The number of mobile users has recently surpassed the number of desktop users, emphasizing the importance of mobile device security. When analyzing how users spend their time on the mobile devices, studies show that it is mostly using mobile apps as opposed to a browser. (see Figure 1).

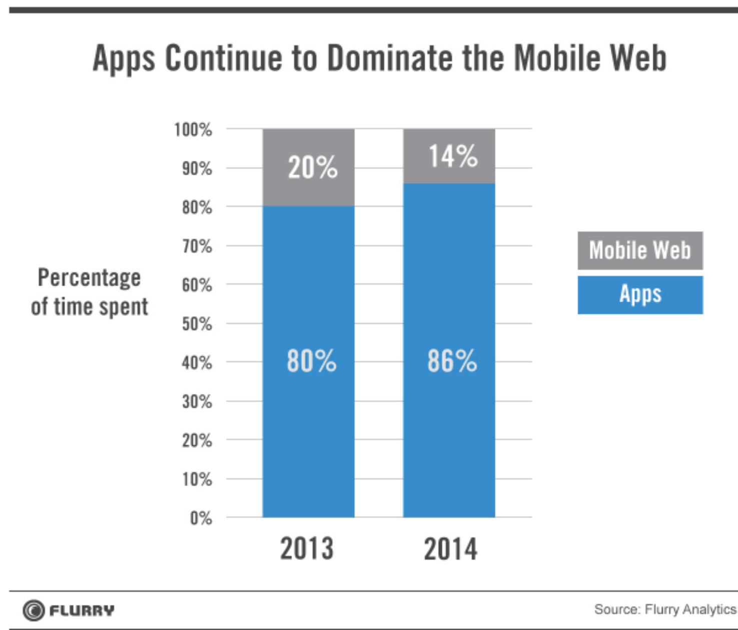

Figure 1: Research shows that users mostly use apps on mobile devices.

Compared to web applications that are generally used with a (desktop) browser, this change in user behavior gives rise to new security challenges. In traditional browser-server applications, data tends to be stored on the server side where tight controls can be enforced. In contrast, many mobile applications cache sensitive data locally on the device, thus exposing it to a number of new attack vectors and often leaving security as the user's responsibility. One main concern is the loss or theft of a device which grants an attacker physical access that may be used to bypass security controls in order to gain access to application data. Depending on the application's data, this can result in a loss of privacy (e.g., health care data, personal pictures and messages) or loss of intellectual property in the case of sensitive corporate data.

In this paper, we first discuss the challenges that arise when storing data locally on a mobile device (see Section section 2) and then describe solutions and limitations for both iOS (Section section 4) and Android (Section section 5).

2 Challenges in Secure Mobile Storage

Mobile devices are used in a number of unique ways that make it challenging to securely store data. Below we look at some of the major challenges in more detail.

2.0.1 Device Mobility

By definition, mobile devices are carried around which makes them prone to theft, loss, and temporary physical access by a malicious actor. A recent study by Consumer Reports [Rep14] showed that during 2013 in the US 1.4 million phones were lost and 3.1 million phones were stolen (both numbers are up from the previous year). A different study from 2012 showed that younger people are more prone to cell phone loss or theft and are thus more likely to have experienced a privacy invasion as a result [BSM12]. For example, 45% of 18-24 year olds reported a lost or stolen phone and 24% reported a privacy invasion. Both numbers decline the older the device owner.

2.0.2 Data Accessibility

Applications and the operating system are required to access data stored on the device not only when the device owner is actively using an application but in many instances also for background operations.

Due to the loss and theft statistics discussed above, data access is difficult to control so a common solution is to protect data using encryption such that obtained data is useless to an attacker. As we will see in more detail in the next sections, basic encryption schemes are insufficient to provide the kind of protection and functionality required on mobile devices.

One core challenge when encrypting data is key storage. On mobile devices the encryption key needs to be available, otherwise the user would not be able to access the data. But securely storing data is the problem we are trying to solve in the first place so simply encrypting data merely shifts the problem from storing the data to storing the key. It should be mentioned that storing an encryption key on a remote server generally does not solve the problem as an attacker who has access to the device can simply request the encryption key from said server and decrypt the data.

Most of today's solutions make use of some form of passcode or password which is used in a cryptographically secure way to derive an encryption key that is then used to encrypt data. The key itself is never stored on the device but only created in memory whenever the user enters their passcode. This brings along a different set of technical and non-technical challenges one of which is usability.

2.0.3 Usability

By their nature, users access mobile devices frequently throughout the day. Implementing security controls such as passcodes or passwords which need to be entered each time the device or an application is used, impedes usability and thus users will opt not to use it. In fact, Consumer Reports published that only 47% of users have a passcode set on their devices with the majority using a four digit code [Rep14].

Now that we introduced the main challenges that secure data storage raises, in the next two sections we discuss how iOS and Android address these challenges. In particular, for each platform we discuss how it supports secure data storage and which APIs are exposed for developers.

3 Threat Model Considerations

When considering security of a system, one should always keep in mind what kind of attacker its data should be protected against. There is a difference in protecting data from remote attackers on the Internet, attackers on the local network, and attackers with physical access. While remote attackers will need to find flaws that are exploitable remotely, a more local attacker may easily gain access to all network communication and may exploit user behavior or flaws to view even TLS-protected traffic. Finally, local attackers may take advantage of non-existent or weak passcodes directly to gain full access to the device.

Similarly, attackers who are concerned about mass-compromising users cannot rely on physical access to devices while targeted attackers can exploit much more intricate situations via social engineering or by obtaining physical access to a device.

Figure 2: Illustration of how defense effort increases with attacker sophistication and capabilities.

In any case, the more sophisticated the attacker and the more capabilities they have, the more effort it takes to protect against them. While it is relatively easy to protect data against mass-compromise using remote attacks (assuming the attacker is not highly capable), security against determined local attackers is disproportionately more difficult (see Figure 2 for an illustration). When thinking about security controls and the trade-off between usability and security it is therefore paramount to have an attacker model in mind which adequately captures the threat one is concerned about. Protecting against all conceivable attacks is not possible. Keep this in mind when reading through the remainder of this paper.

4 Secure Data Storage on iOS

Apple appears to have invested a lot of thought and effort into designing secure storage facilities for iOS. What sets Apple apart from Android (and to some degree Windows Phone) is that they have full control over the entire device ecosystem including hardware and software. As we will see below, this allows them to provide hardware support for the secure storage functions provided by the operating system. Moreover, Apple has thus far not allowed carriers to provide customized iOS versions for their devices reducing the time until operating system updates are available to consumers. Figure 3 shows data provided by Mixpanel [Mix15b] which illustrates that iOS users tend to upgrade soon after a new iOS version is released which results in a fairly homogeneous iOS version deployment (see Figure 4).

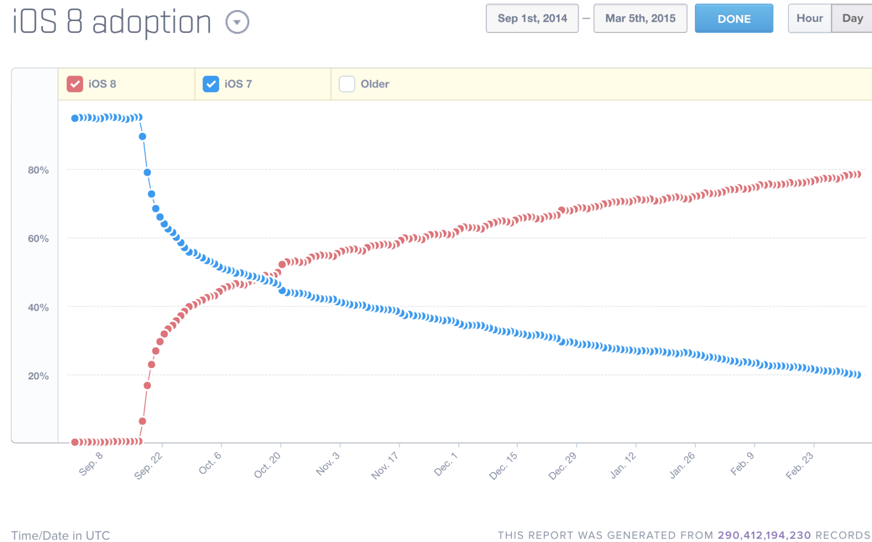

Figure 3: iOS 8 Adoption Rate by Mixpanel [Mix15b]

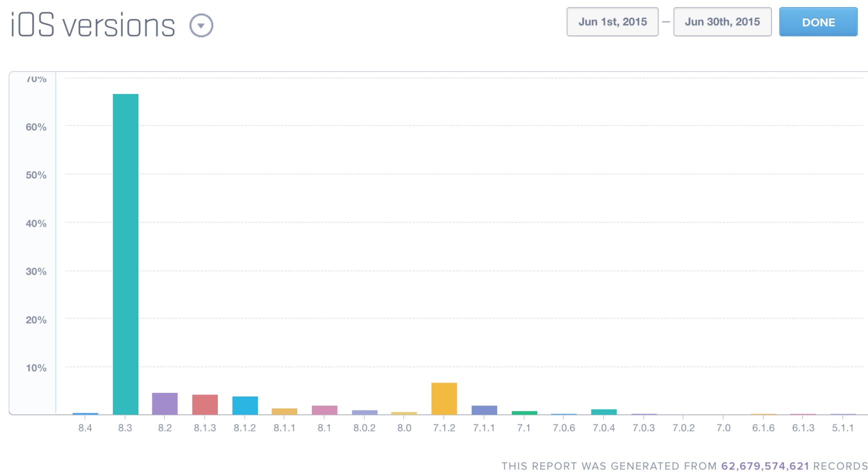

Figure 4: iOS Version Fragmentation by Mixpanel [Mix15c]

The majority of information presented in this section is based on Apple's official iOS security guide [App15b] and additional details were taken from the iOS Hacker's Handbook [MBD+12] or are based on NCC's research. It should be mentioned that due to the release of new iOS versions and iDevice models, some information in the iOS Hacker's Handbook has now been superseded.

4.1 Fundamentals of iOS Data Protection

Apple refers to its facilities for secure data storage as ''Data Protection'' and it is a system which includes a number of moving parts. To give some background on how secure data storage can be implemented from the ground up, our discussion will briefly cover the low-level components and lead up the stack to the public APIs exposed to developers.

4.1.1 Encryption Baked Into Hardware

Using encryption on mobile devices shifts the secure data storage problem from the data to the encryption key. One way to solve this is to provide hardware support which stores an encryption key in a manner that cannot be read from software. Apple chose to embed a unique device identifier, which is used to derive an AES 256-bit key, into the iDevice's SOC (System on a Chip) during manufacturing. According to [App15b], this ID is unique per device and it is not ''recorded by Apple or any of its suppliers''. While the key cannot directly be read, it can be used in order to encrypt and decrypt data similar to the function Trusted Platform Modules [Wik15] provide. Given that only the hardware chip is privy to the device identifier, this mechanism effectively ties any data encrypted using it to the physical device.

In the next section we will see how additional encryption keys are used to provide a number of security guarantees for different accessibility requirements. Those additional keys are generally generated using a Pseudorandom Number Generator (PRG). Apple uses *CTR_DRBG* which is essentially a block cipher such as AES in CTR (Counter) mode. More recent iDevices which have a dedicated ''Secure Enclave'' hardware component make use of hardware random number generators based on ring oscillators [SMS07].

In order to effectively delete encrypted data stored on the device, iOS is able to delete the encryption keys. To do so reliably on modern Solid State Disks (SSDs), the hardware is able to bypass any wear-leveling and access the SSD directly.

Armed with the knowledge of the low-level anchor for iOS encryption, the next section describes how this is used to establish different levels of data protection.

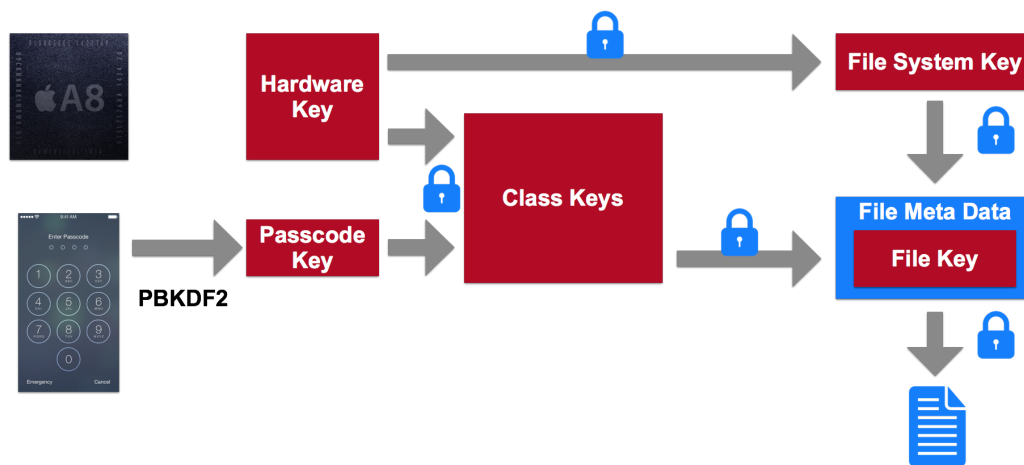

Figure 5: Simplified representation of how the hardware root and passcode integrate with iOS' encryption model.

4.1.2 Encryption Hierarchy

iOS employs a sophisticated hierarchy of keys that are used to encrypt files and Figure 5 shows the fundamental layout of this hierarchy. First, all files stored on the iOS data partition are encrypted[1] using a *file key*. This key is stored as part of the file's meta data. By itself, that is not particularly useful as anyone could simply read the file key and decrypt the file. The individual file meta data and file keys are therefore wrapped[2] using a so-called *file system key*.

The file system key is generated upon iOS installation and it is changed whenever the device is restored. Moreover, it provides a simple means to wipe the entire device since deleting this key will render all data on the device inaccessible. In order to protect the file system key (and all files encrypted with it), the file system key is wrapped using the device-specific hardware key based on the device identifier.

Given this key setup alone, all data on the device file system is encrypted using keys that are tied to the device. That means that an attacker who has physical access to a powered-off device and dumps the flash contents, is unable to decrypt the data offline. However, when the device is turned on, the respective keys are in memory and the operating system decrypts files automatically on the fly. This is where the lower portion of Figure 5 comes into play.

Above the discussion was slightly simplified in that the file system key does not directly wrap the file meta data. There is an additional step in which the file metadata is first wrapped using a so-called data protection class key and the resulting ciphertext is protected by the file system key. The central idea is that the different protection class keys are available under different conditions, e.g., when the device is on or when it has been unlocked. The details on the class keys are discussed in Section section 4.2 along with the additional security guarantees they provide.

4.2 Filesystem Encryption

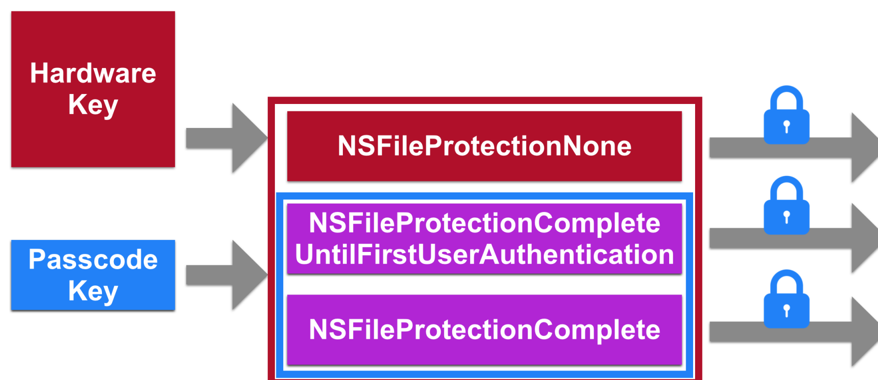

Figure 6: Illustration how the different class keys relate to passcode and hardware key.

In order to store data such that it cannot be recovered by an attacker who has full physical access to the device, there must be some information that is inaccessible to the attacker. In order to achieve this level of protection, iOS incorporates the iOS passcode into the encryption hierarchy (see Figure 6) [App15b]. By combining the passcode, hardware key, and policies defining when to delete an encryption key from memory, iOS is able to provide a number of different *Data Protection Classes* as listed in Table 1 [App15b]. Before discussing the different classes in detail, the next section briefly discusses the role of the passcode in this process.

[1] AES in CBC Mode
[2] Technically, keys are protected using NIST AES key wrapping, per RFC 3394 [SH02].

Protection Class	Meaning
`NSFileProtectionComplete`	Protected when device is locked.
`NSFileProtectionCompleteUnlessOpen`	If the file is open, it can still be read even when the device is locked.
`NSFileProtectionCompleteUntilFirstUserAuthentication`	Protected from boot until user unlocks.
`NSFileProtectionNone`	No effective protection.

Table 1: Overview of iOS file protection classes.

4.2.1 Passcode

Whenever the user unlocks the device by entering the passcode, a cryptographic key is derived from the entered passcode using the Password-Based Key Derivation Function 2 (PBKDF2). The derived key can then be used to unwrap (decrypt) other keys as shown in Figure 6. As hinted on above, in order to prevent an attacker from copying encrypted files off the device and then performing an offline attack on the passcode, the passcode alone cannot be used to recover the protection class keys. The hardware key is also required. Since this key cannot be read by the operating system, any brute-force attack will need to be performed on the device.

Requiring that the attack is performed on the device, grants the device certain abilities which would otherwise not be possible. For instance, the iteration count for PBKDF2 was tuned to require 80 milliseconds for deriving a key which would require 5.5 years to fully brute-force a six-character alphanumeric passcode. However, due to usability and convenience reasons, most users do not choose such a complex passcode [Rep14] even though it would significantly improve the security of their data on the device. Apple attempted to address this usability / security trade-off by introducing TouchID which will be discussed in Section section 4.4. On devices which have a modern processor such as A7 or later, the key derivation is performed by the Secure Enclave. In addition to the iteration count mentioned above, this hardware component also enforces a 5-second delay between unlocking attempts further hardening the device from brute-force attempts.

However, even with a 5 second delay a 4 digit pincode can be brute-forced in 14 hours. This is one of the reasons why Apple has increased the numeric passcode lengths to 6 digits in iOS 9. This pushes the brute-force time to approximately 57 days. It should be noted that in addition to these hardware controls, the iOS software layer is enforcing an exponential back-off for passcode attempts. However this may potentially be bypassed if an attacker has access to an exploit which allows them to modify the underlying iOS code. Hardware controls such as the ones enforced by the Secure Enclave are generally significantly harder to bypass.

Finally, the hardware control of the key derivation also allows the device to enforce a wipe of all data by deleting the file system encryption key if the passcode has been entered incorrectly 10 times, if the user chooses to enable this feature. There was an interesting flaw (CVE-2014-4451) related to this feature which allowed an infite number of passcode attempts by using a hardware device which would forcefully cut the power of the device after each attempt. This would result in the failed attempt to not be recorded or acted upon by the iDevice. However, this increased the time for a single attempt to about 40 seconds [Che15, Rya15]. This flaw has since been fixed in the latest version of iOS [App15a].

4.2.2 Description of Data Protection Classes

As discussed above, iOS automatically encrypts the entire user file system using a device and file-specific key. This is the only protection present when the file uses the `NSFileProtectionNone` class. When this mode is used,

iOS transparently decrypts files for any read operation regardless of whether the device is locked. Since iOS 8, the default protection class was changed such that the decryption key is only available as long as the device is turned on and the user unlocked the device at least once since boot-up (`NSFileProtectionCompleteUntilFirstUserAuthentication`) [App15b]. On reboot or shutdown, the key is deleted and will only be available once the user unlocks the device again. Note that since the key is derived from the user's passcode, this only provides additional protection if the user has in-fact a passcode set. Application developers can opt to use a more tight protection class which uses a key that is also derived from the passcode but which is removed from the device as soon as the user locks[3] the device (`NSFileProtectionComplete`). Additional details on iOS data protection can be found in [MBD+12] and [App15b]. While each of these protection classes can be set on a per-file basis, app developers can choose to define a default protection class via app entitlements which will be used for all files stored by the app unless specified otherwise.

4.3 Keychain

The iOS keychain is a suitable tool for storing smaller secrets in a structured store and it is appropriate to be used when not entire files have to be stored. Internally, the keychain lives in a sqlite database file with each entry being encrypted using a key which according to the item's data protection class.

The keychain provides similar protections options as the ones available for the file system discussed in the previous section (see Table 2) [App15c]. There is one keychain variant for each of the file protection classes discussed above. However, since iOS 8, there is one additional class called `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` that is unique to the keychain. The class essentially operates like the `WhenUnlocked` class with the difference that it only stores a keychain item, if the user's device has a passcode set. This is a desirable feature since, as mentioned previously, a passcode is required in order for `kSecAttrAccessibleWhenUnlocked` and `kSecAttrAccessibleAfterFirstUnlock` to provide additional benefits. This new class ensures that data is not left unprotected should the user choose to not have a passcode. Moreover, it renders the data inaccessible whenever the user removes the passcode from the device.

One important fact to note about the keychain is that keychain data is not deleted when an app is uninstalled. From a usability perspective this allows users to re-install the app at a later time and have the keychain data available for use. However, it also exposes store data in a subtle way should the device be compromised after an app had been uninstalled.

Protection Class	Meaning
`kSecAttrAccessibleWhenUnlocked`	Protected when device is locked.
`kSecAttrAccessibleAfterFirstUnlock`	Protected from boot until user unlocks.
`kSecAttrAccessibleAlways`	No protection.
`kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly`	Store data only when passcode set.

Table 2: Overview of iOS Keychain protection classes.

4.3.1 Keychain Access Control

Compared to plain file encryption, the keychain provides a number of advanced features. Using so-called keychain access groups it is possible for multiple applications by the same developer to share access to keychain items. For this, the applications must share a common app prefix which is possible when the

[3] 10 seconds after the "Require Passcode" setting

applications are published under the same iOS developer program account. Apple enforces the prefix to be unique via code signing, provisioning profiles, as well as software controls. An application can opt to make data which it stores in the keychain accessible by other applications which share its app prefix. This may be desirable for credentials which are shared by two apps from the same developer.

The second feature worth noting is related to further controlling access to keychain items. Just like the different data protection classes added more fine-grained control compared to simple full-disk encryption which leaves data either accessible (when the device is on) or inaccessible (when the device is off), `kSecAttrAccessible-WhenUnlocked` discussed previously provides a similar binary state based on the lock status. Using keychain access control, it is possible to require ''user presence'' when a keychain item is accessed, granting even more fine-grained control as to when data can be access [Inc15]. User presence is established by requiring the user to either enter their passcode or swipe their finger whenever the keychain item is accessed or modified. For iOS 9, it will be possible to specify whether the passcode or the fingerprint are required while older iOS versions will accept either of the two without the application being able to tell which has been used. Moreover, the new API allows the devleoper to detect when the fingerprints enrolled on the device have changed which can be used to render data inaccessible in this case.

4.4 Touch ID

Touch ID was a major step forward for usability on mobile devices but its purpose was frequently misunderstood or misrepresented. While biometrics are often displayed as top security controls, this was not the purpose when adding a fingerprint reader to a phone. Previously, users had to enter a passcode each time they used the device, if they even chose to have this security control in the first place. This led to the majority of users either not using a passcode at all, or choosing one which is fast and easy to enter on a mobile device. With Touch ID, there is no need to enter the passcode throughout the day, making it more attractive to a wider range of users. Moreover, by integrating Touch ID with other features such as Apple Pay and the App Store, users are incentivised to actually enroll their fingerprints and use Touch ID, increasing the overall number of users protecting their phones. While it is true that fingerprints can be lifted and Touch ID can be bypassed like most mainstream fingerprint readers, one could argue that such targeted attacks are not of concern for the majority of the user base. In contrast, not having the device protected and all data readily accessible when it gets stolen or lost is a risk many users are affected by. With that, we dive a bit deeper into how Touch ID works.

When considering the device unlock function of Touch ID, it essentially is a convenience wrapper around the existing technologies discussed above. Touch ID requires the user to set a passcode on the device which is used as part of the encryption hierarchy as before. This is also the reason why the passcode has to be entered the first time one boots the device. Following this first unlock, the key derived from the passcode is protected by Touch ID within the Secure Enclave. Essentially, the key is safeguarded by the Touch ID subsystem and only released when an appropriate finger is swiped. Since data within the secure enclave should be inaccessible from the operating system, it is very difficult for an attacker to gain access to the key on a locked device.

Since the encryption key remains to be based off of the passcode, the level of data security still depends on the passcode as well. Even though users do not need to enter the passcode often anymore, there is no obvious notification or notice for users to choose a strong passcode when they enroll in Touch ID. This could be improved to make Touch ID's usability improvements even more effective.

4.4.1 User Presence vs. Local Authentication

In addition to device unlock, Touch ID is also integrated into the keychain, as mentioned in the previous section. This integration makes use of the Secure Enclave to protect data stored in the keychain.

In contrast to the keychain integration, Local Authentication is an iOS feature which exposes the Touch ID functionality to app developers. They essentially can call an API and ask iOS to verify the user's identity. The API allows the developer to specify whether the user has to use their fingerprint or if the passcode is also an accepted criterion. In contrast to the user presence feature, there is no cryptographic wrapping of data under the fingerprint/data. As a result, if an attacker is able to jailbreak the device, they can override the API functions provided by iOS and bypass the Local Authentication checks [Dou14]. Therefore, keychain storage with user presence appears to be a strictly stronger control than Local Authentication.

Since attacks against the underlying encryption key are likely difficult, and attacker may favor to either forge fingerprints or brute force the passcode. A common concern regarding User Presence is that passcodes may be easier to break than fingerprints are to forge. This raised concerns since, pre iOS 9, for keychain access, there has been no way of specifying whether the user should provide the fingerprint or the passcode (or either one). It is tempting to fall back to Local Authentication in these situation since it grants the desired control. However, the problem with this approach is that if a compromisable passcode is the risk to protect against, an attacker can simply jailbreak the device, override the Local Authentication API, and is still able to access whatever data was protected by Local Authentication without having the correct fingerprint.[4]

4.5 Jailbreaking

Jailbreaking refers to exploiting flaws in iOS to gain root-level access to the device. This is frequently used to bypass the native protections of iOS in order to customize the operating system beyond what Apple allows or to sideload apps. In addition, jailbreaking enables security researchers to gain a better understanding of how iOS and iOS apps work and provides them the insights needed to find security flaws more effectively.

While there are legitimate use-caes for jailbreaking (e.g., customizations, security research), it does not come without dangers for end users. First, many OS-level protections of iOS are purposefully disabled in order to execute arbitrary, non-signed code. Moroeover, having root-level code execution enables anyone with this kind of access to fully control any applications running on the device. If an attacker is able to jailbreak the device, they can exploit this access to gain access to the majority of data stored on the device.

There are, however, some limitations to what jailbreaking can do. First, publicly available jailbreaks can only be used if the device does not have a passcode set. Note that there may well be ways for obtaining root-level access to a device by using non-public exploits which do not require knowledge of the passcode. That said, even if this were to happen, data encrypted under the iOS passcode is not accessible when the device is locked. The caveat to this is that jailbreaking allows the installation of a backdoor on the device which can then be used to exfiltrate data as soon as the user unlocks the device and makes data accesible. In the absence of a remote jailbreak, the process for backdooring may require the attacker to first gain physical access to the device, plant the backdoor, and return the device to the user.

While Apple has been quickly patching security flaws used in public jailbreaks, most version of iOS have seen public jailbreaks sooner or later. The ability for a sufficiently motivated and capable attacker to jailbreak a device should be considered, in conjunction wih the threat model.

[4]Note that after a jailbreak an attacker most likely can simply read the data off the device without bypassing Local Authentication. This is due to the fact that data is not actually protected by encryption but Local Authentication is simply an access control mechanism.

5 Secure Data Storage on Android

5.1 Fundamentals of Android's Secure Data Storage

Google's available data protection has been an evolution starting from the largely unreleased Android 3.0 Honeycomb and reaching a much more mature state with the release of Android 5.x Lollipop. When discussing secure storage on Android, it is relevant to mention the current state of updates to the Android operating system and the varying degrees of protection provided by the myriad different OEM partners producing Android devices.

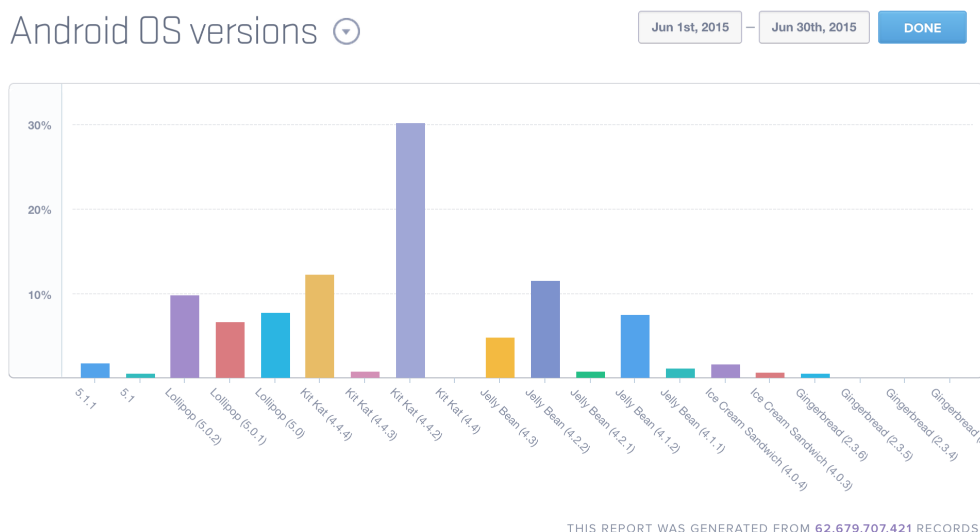

Figure 7: Android versions in use as seen by Mixpanel [Mix15a]

There are many facets to the software update problem Android devices face but the core of the matter is that many devices in the wild can get abandoned on old, vulnerable versions of Android. Complicating matters more, many OEMs provide vastly different bootloader protections such that it's possible on many devices to still write to areas not protected by Google's userdata encryption with physical access to the device. Presented here is a breakdown of the security offered by Android's last few modern releases. This information is from official documentation, NCC's own research, and by other prominent security researchers in the field. Due to various implementation differences from Google's OEM partners, only the implementation provided by Google directly will be discussed in this paper.

5.2 Full-Disk Encryption

At its core, Android's encryption exclusively protects the userdata partition of the device and is based on dm-crypt, the block based disk encryption subsystem of the Linux kernel's device mapper system. Because dm-crypt works at the block level, Android devices with NAND based flash memory using the YAFFS or YAFFS2 (Yet Another Flash FileSystem) cannot leverage this protection. Google uses AES 128-bit in CBC Mode along with ESSIV:SHA256 for initialization vector generation to generate the device encryption key (DEK) or "master key." Individual disk sectors are then encrypted with this master key on the userdata partition [Goo15].

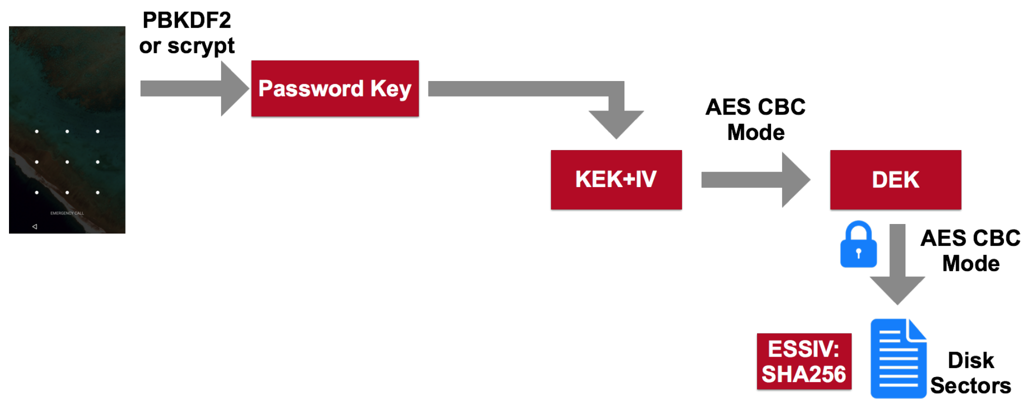

Figure 8: Simplified Android Disk Encryption Flow. (Pre-Android 5.0)

A password based key derivation function is applied to the user-supplied lockscreen code (PIN or Password) and the stored 16-byte salt that was generated when the master key was created. The resulting key is referred to as the Key Encryption Key (KEK) and is used to encrypt the master key. Using this method prevents the necessity for re-encryption of the sectors protected by the master key if a user changes their lockscreen code. In Android 3.0 - 4.3, PBKDF2 was used for KEK generation, but this proved to be a weak security mechanism for several reasons. Users tend to choose short PIN codes or quickly typed passwords to reduce the burden involved with typing their passcode out each time they want to unlock and use their device. This combined with the ease of computing PBKDF2 hashes on modern GPUs allowed easy recovery of keys anywhere from seconds to hours depending on the strength of the user passcode.

To improve this situation, Google introduced scrypt as a method for key derivation in Android KitKat 4.4. This change increases the time required per attempt and also reduces the effectiveness of GPUs for brute forcing due to the memory requirements of scrypt. While this is an improvement, weak passcodes are still very recoverable with sufficient time. [Ele14]

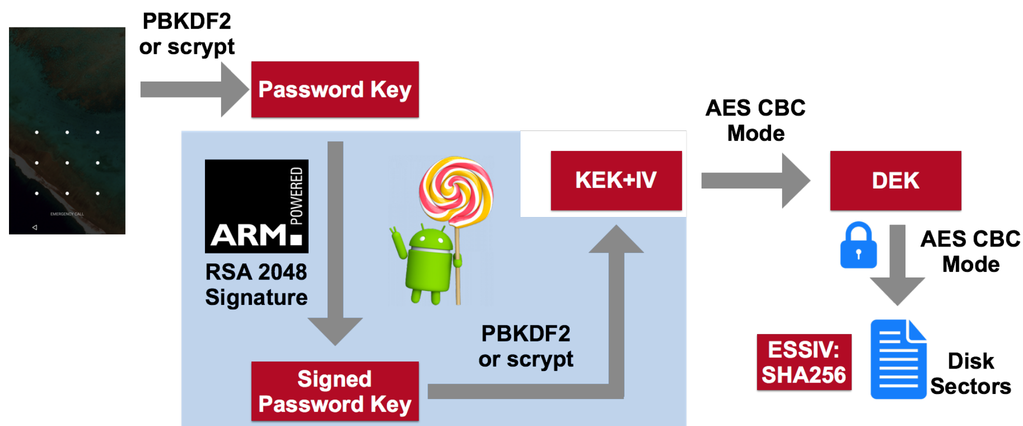

Figure 9: Simplified Android Disk Encryption Flow. (Android 5.0+)

Google improved their full disk encryption further with the release of Android Lollipop 5.x by implementing a few additional features. Most importantly, the Key Encryption Key can now be signed by a key in the device's Trusted Execution Environment (TEE). This effectively removes the ability to perform attacks offline, as the Key Encryption Key now requires this non-removable key from the TEE on the device as part of the decryption process. Additionally, Google attempted to make the optional device encryption protection part of the default

out of box experience with the use of `forceencrypt` for the userdata partition. Unfortunately, this move proved to be problematic for their OEM partners in practice and Google quietly revised this requirement for Lollipop. It is assumed that `forceencrypt` will once again be a requirement in the upcoming "M" release however. Lastly, Android now allows for the use of the Pattern lockscreen type for volume encryption.

5.3 Android Credential Storage

Android's built-in credential storage, like other security features, has grown steadily more secure and robust with each new edition of the operating system. At its core, the system credential store allows for the storage of keys for VPNs, WiFi or from user-installed applications. These keys are then encrypted by a master key derived from a user's lockscreen code before being committed to disk. Asymmetric key pairs can be stored in the system credential store for direct encryption operations, or alternatively they can encrypt a symmetric key which can then encrypt whatever application data a developer wants to keep safe. In modern devices, the credential store can be hardware-backed such that private key material is non-extractable, even as root. This increases the complexity of an attack and requires the use of the specific device directly as part of the process. The KeyStore, which is responsible for maintaining cryptographic keys and their associations with owners, allows applications to create application-specific keys as well as user-specific keys in the case of a multi-user environment.

The system is not without some shortcomings however. For example, the master key is not tied to the device like it is on iOS, so with elevated access, extracting all the encrypted blobs and brute forcing them on much more powerful hardware is possible depending on the specific scenario (OS version, hardware available). Inconsistencies in hardware-backed credential storage being available on various OEM devices is also an issue in that not all devices have equal protection capabilities. Google provides an API to check whether a system has a hardware-backed credential storage, but this means application developers have conditions to check before making decisions about how to store data. In many cases this results in at least three or four different conditions to account for different versions of the operating system.

Furthermore, many application developers and end-users have expressed frustration with the usability of the implementation. Many developers and users have reported bugs where the KeyStore wipes all the stored keys when a user changes from one lockscreen type to another. Numerous bugs have been filed where this behavior is different among versions of Android, or even inconsistencies within the same system and seeing different results. Google has acknowledged and fixed this unpredictable KeyStore behavior in Android Lollipop 5.0 [Cod13, Cod14], but as discussed earlier, the usage rate for the newest versions of Android is not representative of the majority of users. According to API documentation for the unreleased Android M reveal that Google has addressed these frustrations with clear documentation and expectations [Goo]. In the authors' interaction with clients during the course of application security assessments, a frequent comment is that the KeyStore cannot be relied on. This perception leads to in-house development of alternative solutions, with varying results. This almost always ends with implementation flaws and solutions which have not been properly audited, creating a dangerous situation for data security.

Looking ahead, Google has added support for symmetric keys in the upcoming "M" release of Android. Prior to this support, users had to access private APIs in order to store arbitrary application secrets. This method has the downside of using undocumented functions and being unable to rely on these features persisting to the next release [Ele12]. Additionally, Google has extended more fine-grain control to users over keys managed by the system and how they should be used.

5.4 Platform Diversity Considerations

The Android operating system runs on a diverse portfolio of devices suiting a wide variety of markets and price points from many different manufacturers. With that kind of diversity, devices come with a wide range of different boot setups and integrity mechanisms. Device configurations tend to vary by OEM partner, but differences can be observed even within devices from the same manufacturer. For example, the same model phone sold via different carriers may have several variations in data protections offered, hardware or internal storage configurations.

Device manufacturers extend and modify Android from what Google originally publishes in the Android Open Source Project (AOSP). This is done for a number of reasons but largely due to brand recognition. Each OEM has their own unique look and feature set they provide in their versions of Android in an attempt to distinguish their brand from the numerous other Android device makers. In addition to UI design, cutting edge hardware and its associated driver software are also present in flagship models. This creates potential security concerns because this proprietary code is largely opaque to the public (including researchers) and has been a frequent source of security problems in the past. Together, these point to development and security review processes which may be somewhat lacking or inconsistent. Overly permissive files and folders, bootloader flaws and vulnerable carrier included applications are a consequence of these custom modifications to the platform.

Each OEM differs in how they protect the integrity of the device and one may conclude that these differences are made at a mobile carrier's request. Most notably, Verizon tends to enforce the tightest restrictions for devices on their networks [HTC15], while carriers like AT&T or Sprint protect select devices. Many OEMs offer a bootloader unlock option for developers and hackers who want to run custom code on their devices. OEMs have historically removed the ability for these official unlocks to work for Verizon devices, requiring the use of much more dangerous modifications in order to provide this level of access to internal storage. Such modifications remove the eMMC write protection of partitions on the internal storage that most users have no need to modify. As a result, device security is weakened and the device is now more susceptible to modification without its owner being aware.

These inconsistencies and variations provide a precarious situation when discussing device security that is difficult to navigate. Many manufacturers have recently begun providing for more consistency among variants of a device they produce, but this only solves a small part of a larger ecosystem issue. Newer security protections do exist but are inconsistently applied or are simply ineffective. The same device that is unlockable on one carrier is blocked from doing so on another for example. Some device makers detect you have modified the stock firmware and issue warnings at boot time, but these can be easily overlooked or misunderstood by users.

5.5 Consequences of Inconsistent Bootloader Security

Android devices come with a variety of different bootloader configurations. Many of the popular configurations allow a user to unlock their device through officially supported mechanisms. These typically take the form of running particular unlock commands via a bootloader interface or using unlock tokens provided through the manufacturer's developer program. Unlocking in this manner responsibly erases all personal data on the device as part of the process and then allows the user to overwrite certain partitions on the internal storage. This ensures that unlocked a user's device can't be used to gain access to their data. Other devices have no unlock capabilities.

Some device makers, however, ship permissive bootloaders by default which have no real concept of locked or unlocked. Such devices allow a user to arbitrarily write firmware via a specific boot mode using specialized tools. Knowledge of these tools and what you can do with them is widely discussed on the Internet in mobile device development and hacking communities. In addition to these tools, files which are either official or

user created are also distributed to accomplish various modifications. Often times, these tools provide access to modify device firmware directly. Due to inconsistencies among various OEMs in enforcing boot image signature checking or providing write protection to sensitive partitions, it is possible to gain complete control over a device in many cases.

For most users who wish to modify their devices, or run custom builds of Android, a custom bootable recovery image is typically loaded onto a device first after bootloader access is achieved. Most Android devices ship with a special boot mode called recovery mode. This boot mode loads a tiny Linux environment and a minimal ramdisk and is responsible for installing carrier or OEM signed Over-The-Air (OTA) updates to the device to upgrade its software. It also provides a means to wipe user data from the device or repair a broken installation of Android. This stock recovery image mode has minimal access and control outside of the installation and factory reset procedures it provides. Shell access is limited and restricted to an account with minimal capabilities.

Custom recovery images provide much more control over the device and are extended with many more useful binaries and features. A custom recovery image provides a platform to back up essential firmware on the device, but it also allows for scriptable installation and automated loading of custom Android software. Most importantly, custom recovery images provide root access to the device and the ability to access any of the device's filesystems not protected by volume encryption.

With this capability, it is easy to see why userdata encryption should always be enabled and the integrity of the boot firmware on the device be enforced. Android's encryption mechanism protects information on the userdata partition, rendering it unreadable even to a custom recovery image. However, other partitions, such as the system and boot partitions, are modifiable on many common devices. Therefore, a malicious actor with physical access can leverage a custom recovery image to modify these partitions, including the Linux kernel, and backdoor the device. This allows for, among other things, encryption key recovery. Once an attacker has the key, they can decrypt the otherwise unreadable userdata partition and recover the full contents of the device.

Enter Rosie, a proof of concept "evil maid" attack that was created to illustrate problems with current secure storage implementations. Since it is possible to replace the bootable firmware on many devices due to inconsistent or non-existent signature checks, it is possible to fully compromise even modern devices. Rosie is directly compiled into a custom kernel that is then loaded onto the device. Once installed, Rosie becomes active on the first boot after the victim's device has been tampered. For the purposes of this proof of concept, it was designed to siphon any file off the device and send its payload via UDP to a cloud hosted server for inspection. Because Rosie runs completely within the kernel, there is no need to modify the core system partition on the device and it has full privileges on the target system. Modifying the system partition is entirely possible on devices without strong chains of trust in their boot configurations, but it has the potential to be more complicated due to various OEM security measures.

Rosie is capable of doing virtually anything an attacker wishes it to do. Passwords, session tokens or other private application data can be accessed, a reverse shell could be implemented that would provide even more control, and the entirety of the kernel's cryptographic functionality can be hooked. Rosie has been used to pull the volume encryption key off of a device the moment the userdata partition is decrypted and mounted. This allows for an offline attack scenario in the case where the entire volume has been copied by an attacker. What's more alarming about the ability to even use something like Rosie in the first place is the speed at which it can be deployed on a device in physical posession of a malicious user. Rosie was tested to be installed in under a minute in most cases, even when the device is completely powered off. This scenario is especially interesting because a device encrypted with full volume encryption is usually thought safer when powered off.

Rosie undermines this protection by attacking the weak areas of a device's bootup trust chain and bypasses the need to execute an attack against a running device as a result.

6 Conclusions

As discussed above, securing data on a mobile device is challenging, especially when keeping usability in mind. Apple, with the distinct advantage of being able to supply hardware and software, can provide a more homogeneous solution which incorporates hardware and software controls. Google has been introducing security features with each new release but their adoption is hindered by the delayed availability of updates, and frequent lack of support for aging devices.

Both platforms are at risk if the underlying operating system can be compromised by an attacker. For iOS this requires a jailbreak while for some Android devices with permissive bootloaders no particular exploit is needed to install a backdoor on the device given even brief access . While powerful attacks, these are likely only applied in targeted attacks which may not be a concern for the majority of users.

The trend set by Apple in protecting data until it is actually needed (keychain user presence) as well as in improving the adoption of security controls by enhancing usability is laudable and innovation in this space is needed to protect the average user going forward.

References

[App15a] Apple. About the security content of iOS 8.1.1. https://support.apple.com/en-us/HT204418, 05 2015. 9

[App15b] Apple Inc. iOS Security - iOS 8.3 or later. https://www.apple.com/privacy/docs/iOS_Security_Guide_Oct_2014.pdf, April 2015. 6, 7, 8, 10

[App15c] Apple Inc. Keychain Services Reference. https://developer.apple.com/library/ios/documentation/Security/Reference/keychainservices/index.html, 2015. 10

[BSM12] Jan Lauren Boyles, Aaron Smith, and Mary Madden. Privacy and data management on mobile devices. *Pew Internet & American Life Project*, 2012. 4

[Che15] Dominic Chell. Apple iOS Hardware Assisted Screenlock Bruteforce. http://blog.mdsec.co.uk/2015/03/bruteforcing-ios-screenlock.html, 03 2015. 9

[Cod13] Google Code. AndroidKeyStore deleted after changing screen lock type. https://code.google.com/p/android/issues/detail?id=61989, November 2013. 15

[Cod14] Google Code. AndroidKeyStore deleted after changing screen lock type or clear credentials. https://code.google.com/p/android/issues/detail?id=83218, December 2014. 15

[Dou14] Grant Douglas. SuccessID - TouchID override & simulation. https://hexplo.it/successid-touchid-override-simulation/, 11 2014. 12

[Ele12] Nikolay Elenkov. Storing Application Secrets in Android's Credential Storage. http://nelenkov.blogspot.com/2012/05/storing-application-secrets-in-androids.html, June 2012. 15

[Ele14] Nikolay Elenkov. Revisiting Android Disk Encryption. http://nelenkov.blogspot.com/2014/10/revisiting-android-disk-encryption.html?view=sidebar, October 2014. 14

[Goo] Google. Android Keystore Changes. https://developer.android.com/preview/behavior-changes.html#behavior-keystore. 15

[Goo15] Google. Android - Encryption. https://source.android.com/devices/tech/security/encryption/, July 2015. 13

[HTC15] HTC. Unlocking Your Bootloader. http://www.htcdev.com/bootloader, 2015. 16

[Inc15] Apple Inc. Security Changes. https://developer.apple.com/library/ios/releasenotes/General/iOS80APIDiffs/frameworks/Security.html, 2015. 11

[MBD+12] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.P. Weinmann. *iOS Hacker's Handbook*. ITPro collection. Wiley, 2012. 6, 10

[Mix15a] Mixpanel. Android OS Versions. https://mixpanel.com/trends/#report/android_frag/from_date:-49,to_date:-20, July 2015. 13

[Mix15b] Mixpanel. iOS 8 Adoption. https://mixpanel.com/trends/#report/ios_8/from_date:-321,report_unit:day,to_date:-136, July 2015. 6

[Mix15c] Mixpanel. iOS Versions. https://mixpanel.com/trends/#report/ios_frag/from_date:-48,to_date:-19, July 2015. 6

[Rep14] Consumer Reports. Smart phone thefts rose to 3.1 million last year, Consumer Reports finds, May 2014. 4, 9

[Rya15] Stuart Ryan. CVE-2014-4451 – Apple iOS bug allowing unlimited incorrect pin attempts. http://technicalnotebook.com/software-bugs/apple-ios-bug-allowing-unlimited-incorrect-pin-attempts/, 11 2015. 9

[SH02] J. Schaad and R. Housley. RFC 3394: Advanced Encryption Standard (AES) Key Wrap Algorithm. *Request for Comments*, 2002. 8

[SMS07] Berk Sunar, William J Martin, and Douglas R Stinson. A provably secure true random number generator with built-in tolerance to active attacks. *Computers, IEEE Transactions on*, 56(1):109–119, 2007. 7

[Wik15] Wikipedia. Trusted Platform Module — Wikipedia, The Free Encyclopedia, 2015. [Online; accessed 27-July-2015]. 7