Data Security on Mobile Devices: Current State of the Art, Open Problems, and Proposed Solutions

Maximilian Zinkus
Johns Hopkins University
zinkus@cs.jhu.edu

Tushar M. Jois
Johns Hopkins University
jois@cs.jhu.edu

Matthew Green
Johns Hopkins University
mgreen@cs.jhu.edu

November 19, 2020

Executive Summary

In this work we present definitive evidence, analysis, and (where needed) speculation to answer the questions, (1) *"Which concrete security measures in mobile devices meaningfully prevent unauthorized access to user data?"* (2) *"In what ways are modern mobile devices accessed by unauthorized parties?"* and finally, (3) *"How can we improve modern mobile devices to prevent unauthorized access?"*

We divide our attention between two major platforms in the mobile space, iOS and Android, and for each we provide a thorough investigation of existing and historical security features, evidence-based discussion of known security bypass techniques, and concrete recommendations for remediation. In iOS we find a powerful and compelling set of security and privacy controls, backed and empowered by strong encryption, and yet a *critical lack in coverage* due to under-utilization of these tools leading to serious privacy and security concerns. In Android we find strong protections emerging in the very latest flagship devices, but simultaneously *fragmented and inconsistent security and privacy controls*, not least due to disconnects between Google and Android phone manufacturers, the deeply lagging rate of Android updates reaching devices, and various software architectural considerations. We also find, in both platforms, exacerbating factors due to increased synchronization of data with cloud services.

The markets for exploits and forensic software tools which target these platforms are alive and well. We aggregate and analyze public records, documentation, articles, and blog postings to categorize and discuss unauthorized bypass of security features by hackers and law enforcement alike. Motivated by an increasing number of cases since Apple v. FBI in 2016, we analyze the concrete impact of forensic tools, and the privacy risks involved in unchecked seizure and search. Then, we provide in-depth analysis of the data potentially accessed via common law enforcement methodologies from both mobile devices and accompanying cloud services.

Our fact-gathering and analysis allow us to make a number recommendations for improving data security on these devices. In both iOS and Android we propose concrete improvements which mitigate or entirely address many concerns we raise, and provide ideation towards resolving the remainder. The mitigations we propose can be largely summarized as the increased coverage of sensitive data via strong encryption, but we detail various challenges and approaches towards this goal and others.

It is our hope that this work stimulates mobile device development and research towards increased security and privacy, promotes understanding as a unique reference of information, and acts as an evidence-based argument for the importance of reliable encryption to privacy,

which we believe is both a human right and integral to a functioning democracy.

NOTE: This document is a draft and may contain technical errors. Please contact the authors if you encounter any inaccurate information.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile devices have become a ubiquitous component of modern life. More than 45% of the global population uses a smartphone [1], while this number exceeds 96% in the United States [2]. This widespread adoption is a double-edged sword: the smartphone vastly improves the amount of information that individuals can carry with them; at the same time, it has created new potential targets for third parties to obtain user data. The portability and ease of access makes smartphones a target for malicious actors and law enforcement alike: to the former, it provides new opportunities for criminality [3, 4, 5]. To the latter it offers new avenues for investigation by law enforcement agencies [6, 7, 8].

Over the past decade, hardware and software manufacturers have acknowledged these concerns, in the process deploying a series of major upgrades to smartphone hardware and operating systems. These include mechanisms designed to improve software security; default use of passcodes and biometric authentication; and the incorporation of strong encryption mechanisms to protect data in motion and at rest. While these improvements have enhanced the ability of smartphones to prevent data theft, they have provoked a backlash from the law enforcement community. This reaction is best exemplified by the FBI's "Going Dark" initiative [8], which seeks to increase law enforcement's access to encrypted data via legislative and policy initiatives. These concerns have also motivated law enforcement agencies, in collaboration with industry partners, to invest in developing and acquiring technical means for bypassing smartphone security features. This dynamic broke into the public consciousness during the 2016 "Apple v. FBI" controversy [9, 10, 11, 12], in which Apple contested an FBI demand to bypass technical security measures. However, a vigorous debate over these issues continues to this day [6, 7, 13, 14, 15, 16]. Since 2015 and in the US alone, hundreds of thousands of forensic searches of mobile devices have been executed by over 2,000 law enforcement agencies, in all 50 states and the District of Columbia, which have purchased tools implementing such bypass measures [17].

The tug-of-war between device manufacturers, law enforcement, and third-party vendors has an important consequence for users: at any given moment, it is difficult to know which smartphone security features are operating as intended, and which can be bypassed via technical means. The resulting confusion is made worse by the fact that manufacturers and law enforcement routinely withhold technical details from the public and from each

other. What limited information is available may be distributed across many different sources, ranging from obscure technical documents to court filings. Moreover, these documents can sometimes embed important technical information that is only meaningful to an expert in the field. Finally, competing interests between law enforcement and manufacturers may result in compromises that negatively affect user security [18].

The outcome of these inconsistent protections and increasing law enforcement access is the creation of massive potential for violations of privacy. More than potential, technology is *already* allowing law enforcement agencies around the world to surveil people [17, 19, 20, 21]. Technological solutions are only part of the path to remediating these issues, and while we leave the policy advocacy and work to experts in those areas, we present these contributions in pursuit of progress on the technical front.

Our contributions. In this work we attempt a full accounting of the current and historical status of smartphone security measures. We focus on several of the most popular device types, and present a complete description of both the available security mechanisms in these devices, as well as a summary of the known *public* information on the state-of-the-art in bypass techniques for each. Our goal is to provide a single periodically updated guide that serves to detail the public state of data security in modern smartphones.

More concretely, we make the following specific contributions:

1. We provide a technical overview of the key data security features included in modern Apple and Android-based smartphones, operating systems (OSes), and cloud backup systems. We discuss which forms of data are available in these systems, and under what scenarios this data is protected. Finally, to provide context for this description, we also offer a historical timeline detailing known improvements in each feature.

2. We analyze more than a decade of public information on software exploits and DHS forensic reports and investigative documents, with the goal of formulating an understanding of which security features are (and historically have been) bypassed by criminals and law enforcement, and which security features are currently operating as designed.

3. Based on the understanding above, we suggest changes and improvements that could further harden smartphones against unauthorized access.

We enter this analysis with two major goals. The first is an attempt to solve a puzzle: despite substantial technological advances in systems that protect user data, law enforcement agencies appear to be accessing device data with increasing sophistication [6, 22, 23]. This implies that law enforcement, at least, has become adept at bypassing these security mechanisms. A major goal in our analysis is to to understand how this access is being conducted, on the theory that any vulnerabilities used by public law enforcement agencies could also be used by malicious actors.

A second and related goal of this analysis is to help provide context for the current debate about law enforcement access to smartphone encrypted data [10, 11, 12, 24], by

demonstrating which classes of data are already accessible to law enforcement today via known technological bypass techniques. We additionally seek to determine which security technologies are effectively securing user data, and which technologies require improvement.

Platforms examined. Our analysis focuses on the two most popular OS platforms currently in use by smartphone vendors: Apple's iOS and Google's Android.[1] We begin by enumerating the key security technologies available in each platform, and then we discuss the development of these technologies over time. Our primary goal in each case is to develop an understanding of the *current state* of the technological protection measures that protect user data on each platform.

Sources of bypass data. Having described these technological mechanisms, we then focus our analysis on known techniques for bypassing smartphone security measures. Much of this information is well-known to those versed in smartphone technology. However, in order to gain a deeper understanding of this area, we also examined a large corpus of forensic test results published by the U.S. Department of Homeland Security, as well as scouring public court documents for evidence of surprising new techniques. This analysis provides us with a complete picture of which security mechanisms are likely to be bypassed, and the impact of such bypasses.

Threat Model. In this work we focus on two sources of device data compromise: (1) physical access to a device, e.g. via device seizure or theft, and (2) remote access to data via cloud services. The physical access scenario assumes that the attacker has gained access to the device, and can physically or logically exploit it via the provided interfaces. Since obtaining data is relatively straightforward when the attacker has authorized access to the device, we focus primarily on unauthorized access scenarios in which the attacker does not possess the passcode or login credentials needed to access the device.

By contrast, our cloud access scenario assumes that the attacker has gained access to cloud-stored data. This access may be obtained through credential theft (e.g. spear-phishing attack), social engineering of cloud provider employees, or via legitimate investigative requests made to cloud providers by law enforcement authorities. While we note that legitimate investigative procedures differ from criminal access from a legal and moral point of view, we group these attacks together due to the fact that they leverage similar technological capabilities.

1.1 Summary of Key Findings

We now provide a list of our key findings for both Apple iOS and Google Android devices.

[1] An important caveat in our Android analysis is that in practice, the Android device-base is significantly more fragmented than Apple's, due to the fact that Android phones are manufactured by many different vendors. As a consequence, our analysis concentrates mainly on "flagship" Google devices and high-end devices that hold a significant degree of market share.

1.1.1 Apple iOS

Apple iOS devices (iPhones, iPads[2]) incorporate a number of security features that are intended to limit unauthorized access to user data. These include software restrictions, biometric access control sensors, and widespread use of encryption within the platform. Apple is also noteworthy for three reasons: (1) the company has overall control of both the hardware and operating system software deployed on its devices, (2) the company's business model closely restricts which software can be installed on the device, and (3) Apple management has, in the past, expressed vocal opposition to making technical changes in their devices that would facilitate law enforcement access [10].[3]

To determine the level of security currently provided by Apple against sophisticated attackers, we considered the full scope of Apple's public documentation, as well as published reports from the U.S. Department of Homeland Security (DHS), postings from mobile forensics companies, and other documents in the public record. Our main findings are as follows:

Limited benefit of encryption for powered-on devices. Apple advertises the broad use of encryption to protect user data stored on-device [27, 28, 29]. However, we observed that a surprising amount of sensitive data maintained by built-in applications is protected using a weak "available after first unlock" (AFU) protection class, which does not evict decryption keys from memory when the phone is locked. The impact is that the vast majority of sensitive user data from Apple's built-in applications can be accessed from a phone that is captured and logically exploited while it is in a powered-on (but locked) state. We also found circumstantial evidence from a 2014 update to Apple's documentation that the company has, in the past, reduced the protection class assurances regarding certain system data, to unknown effect.

Finally, we found circumstantial evidence in both the DHS procedures and investigative documents that law enforcement now routinely exploits the availability of decryption keys to capture large amounts of sensitive data from locked phones. Documents acquired by Upturn, a privacy advocate organization, support these conclusions, documenting law enforcement records of passcode recovery against both powered-off and simply locked iPhones of all generations [30].

Weaknesses of cloud backup and services. Apple's iCloud service provides cloud-based device backup and real-time synchronization features. By default, this includes photos, email, contacts, calendars, reminders, notes, text messages (iMessage and SMS/MMS), Safari data (bookmarks, search and browsing history), Apple Home[4] data, Game Center[5] data, and cloud storage for installed apps.

[2]Although our focus in this work is on smartphones, we mention other devices that share a common security platform.

[3]It should be noted that in practice, Apple has assisted law enforcement operations as required by law, rather than filing suit [25], and continue to provide data to U.S. and other law enforcement agencies upon request at increasing rates, per the Apple Transparency Report [26].

[4]Apple Home provides integrations for Internet of Things devices.

[5]Game Center is Apple's built-in gaming social media service.

We examine the current state of data protection for iCloud, and determine (unsurprisingly) that activation of these features transmits an abundance of user data to Apple's servers, in a form that can be accessed remotely by criminals who gain unauthorized access to a user's cloud account, as well as authorized law enforcement agencies with subpoena power. More surprisingly, we identify several counter-intuitive features of iCloud that increase the vulnerability of this system. As one example, Apple's "Messages in iCloud" feature advertises the use of an Apple-inaccessible "end-to-end" encrypted container for synchronizing messages across devices [31]. However, activation of iCloud Backup in tandem causes the decryption key for this container to be uploaded to Apple's servers in a form that Apple (and potential attackers, or law enforcement) can access [31]. Similarly, we observe that Apple's iCloud Backup design results in the transmission of device-specific *file encryption keys* to Apple. Since these keys are the same keys used to encrypt data on the device, this transmission may pose a risk in the event that a device is subsequently physically compromised.[6]

More generally, we find that the documentation and user interface of these backup and synchronization features are confusing, and may lead to users unintentionally transmitting certain classes of data to Apple's servers.

Evidence of past hardware (SEP) compromise. iOS devices place strict limits on passcode guessing attacks[7] through the assistance of a dedicated processor known as the Secure Enclave processor (SEP). We examined the public investigative record to review evidence that strongly indicates that as of 2018, passcode guessing attacks were feasible on SEP-enabled iPhones using a tool called GrayKey. To our knowledge, this most likely indicates that a software bypass of the SEP was available in-the-wild during this timeframe. We also reviewed more recent public evidence, and were not able to find dispositive evidence that this exploit is still in use for more recent phones (or whether exploits still exist for older iPhones). Given how critical the SEP is to the ongoing security of the iPhone product line, we flag this uncertainty as a serious risk to consumers.

Limitations of "end-to-end encrypted" cloud services. Several Apple iCloud services advertise "end-to-end" encryption in which only the user (with knowledge of a password or passcode) can access cloud-stored data. These services are optionally provided in Apple's CloudKit containers and via the iCloud Keychain backup service. Implementation of this feature is accomplished via the use of dedicated Hardware Security Modules (HSMs) provisioned at Apple's data centers. These devices store encryption keys in a form that can only be accessed by a user, and are programmed by Apple such that cloud service operators cannot transfer information out of an HSM without user permission [27, 28].

[6] Particularly, the iCloud Backup Keybag encrypts file keys with asymmetric class keys [27, 28]. Some encrypted data is accessible to Apple [31], and other data is "end-to-end encrypted."

[7] This describes an attack in which the attacker tests many (or all) possible passcodes in order to identify the user passcode and thus derive encryption keys.

As noted above, our finding is that the end-to-end confidentiality of some encrypted services is undermined when used in tandem with the iCloud backup service. More critically, we observe that Apple's documentation and user settings blur the distinction between "encrypted" (such that Apple has access) and "end-to-end encrypted" in a manner that makes it difficult to understand which data is available to Apple. Finally, we observe a fundamental weakness in the system: Apple can easily cause user data to be re-provisioned to a new (and possibly compromised) HSM simply by presenting a single dialog on a user's phone. We discuss techniques for mitigating this vulnerability.

Based on these findings, our overall conclusion is that data for iOS devices is highly available to both sophisticated criminals and law enforcement actors with either cloud or physical access. This is due to a combination of the weak protections offered by current Apple iCloud services, and weak defaults used for encrypting sensitive user data on-device. The impact of these choices is that Apple's data protection is *fragile*: once certain software or cloud authentication features are breached, attackers can access the vast majority of sensitive user data on device. Later in this work we propose improvements aimed at improving the resilience of Apple's security measures.

1.1.2 Google Android & Android Phones

Google's Android operating system, and many third-party phones that use Android, incorporates a number of security features that are analogous to those provided by Apple devices. Unlike Apple, Google does not fully control the hardware and software stack on all Android-compatible smartphones: some Google Android phones are manufactured entirely by Google, while other devices are manufactured by third parties. Moreover, device manufacturers routinely modify the Android operating system prior to deployment.

This fact makes a complete analysis of the Android smartphone ecosystem more challenging. In this work, we choose to focus on a number of high-profile phones such as Google Pixel devices and recent-model Samsung Galaxy phones, mainly because these devices are either (1) representative devices designed by Google to fully encapsulate the capabilities of the Android OS, or (2) best-selling Android phones, with large numbers of active devices worldwide. We additionally focus primarily on recent versions of Android (Android 10 and 11, as of this writing). We note, however, that the Android ecosystem is highly fragmented, and contains large numbers of older-model phones that no longer receive OS software updates, a diversity of manufacturers, and a subclass of phones which are built using inexpensive hardware that lacks advanced security capabilities. Our findings in this analysis are therefore necessarily incomplete, and should be viewed as an optimistic "best case."

To determine the level of security currently provided by these Android devices against sophisticated attackers, we considered the full scope of Google's public documentation, as well as published reports from the U.S. Department of Homeland Security (DHS), postings from mobile forensics companies, and other documents in the public record. Our main findings are as follows:

Limited benefit of encryption for powered-on devices. Like Apple iOS, Google Android provides encryption for files and data stored on disk. However, Android's encryption mechanisms provide fewer gradations of protection. In particular, Android provides no equivalent of Apple's *Complete Protection* (CP) encryption class, which evicts decryption keys from memory shortly after the phone is locked. As a consequence, Android decryption keys remain in memory at all times after "first unlock," and user data is potentially vulnerable to forensic capture.

De-prioritization of end-to-end encrypted backup. Android incorporates an end-to-end encrypted backup service based on physical hardware devices stored on Google's datacenters. The design of this system ensures that recovery of backups can only occur if initiated by a user who knows the backup passcode, an on-device key protected by the user's PIN or other authentication factor. Unfortunately, the end-to-end encrypted backup service must be opted-in to by app developers, and is paralleled by the opt-out Android Auto-Backup, which simply synchronizes app data to Google Drive, encrypted with keys held by Google.

Large attack surface. Android is the composition of systems developed by various organizations and companies. The Android kernel has Linux at its core, but also contains chip vendor- and device manufacturer-specific modification. Apps, along with support libraries, integrate with system components and provide their own services to the rest of the device. Because the development of these components is not centralized, cohesively integrating security for all of Android would require significant coordination, and in many cases such efforts are lacking or nonexistent.

Limited use of end-to-end encryption. End-to-end encryption for messages in Android is only provided by default in third-party messaging applications. Native Android applications do not provide end-to-end encryption: the only exception being Google Duo, which provides end-to-end encrypted video calls. This lack of default end-to-end encryption for messages allows the service provider (for example, Google) to view messages and logs, potentially putting user data at risk from hacking, unwanted targeted advertising, subpoena, and surveillance systems. End-to-end encrypted messaging over RCS[8] may be in development for Android [32].

Availability of data in services. Android has deep integration with Google services, such as Drive, Gmail, and Photos. Android phones that utilize these services (the large majority of them [33, 34]) send data to Google, which stores the data under keys it controls - effectively an extension of the lack of end-to-end encryption beyond just messaging services. These services accumulate rich sets of information on users that can be exfiltrated either by knowledgeable criminals (via system compromise) or by law enforcement (via subpoena power).

[8]Rich Communication Services, a cell network carrier replacement protocol for SMS and MMS.

Chapter 2

Technical Background

2.1 Data Security Technologies for Mobile Devices

Modern smartphones generate and store immense amounts of sensitive personal information. This data comes in many forms, including photographs, text messages, emails, location data, health information, biometric templates, web browsing history, social media records, passwords, and other documents. Access control for this data is maintained via several essential technologies, which we describe below.

Software security and isolation. Modern smartphone operating systems are designed to enforce access control for users and application software. This includes restricting input/output access to the device, as well as ensuring that malicious applications cannot access data to which they are not entitled. Bypassing these restrictions to run arbitrary code requires making fundamental changes to the operating system, either in memory or on disk, a technique that is sometimes called "jailbreaking" on iOS or "rooting" on Android.

Passcodes and biometric access. Access to an Apple or Android smartphone's user interface is, in a default installation, gated by a user-selected passcode of arbitrary strength. Many devices also deploy biometric sensors based on fingerprint or face-recognition as an alternative means to unlock the device.

Disk and file encryption. Smartphone operating systems embed data encryption at either the file or disk volume level to protect access to files. This enforces access control to data even in cases where an attacker has bypassed the software mechanisms controlling access to the device. Encryption mechanisms typically derive encryption keys as a function of the user-selected passcode and device-embedded secrets, which is designed to ensure that access to the device requires both user consent and physical control of the hardware.

Secure device hardware. Increasingly, smartphone manufacturers have begun to deploy secure co-processors and virtualized equivalents[1] in order to harden devices against both software attacks and physical attacks on device hardware. These devices are designed to strengthen the encryption mechanisms, and to guard critical data such as biometric templates.

Secure backup and cloud systems. Most smartphone operating systems offer cloud-based data backup, as well as real-time cloud services for sharing information with other devices. Historically, access to cloud backups has been gated by access controls that are solely under the discretion of the cloud service provider,such as password authentication, making this a fruitful target for both attackers and law enforcement to gain access to data. More recently, providers have begun to deploy provider-inaccessible encrypted backup systems that enforce access controls that require user-selected passcodes, with security of the data enforced by trusted hardware at the providers' premises.

2.2 Threat Model

In order to discuss the vulnerability of mobile devices it is pertinent to consider the threat models which underpin our analysis. In fact, in this case the threats of the traditional remote network adversary are relatively well-mitigated. It is with a particular additional capability that our threat actors, namely law enforcement forensic investigators and criminal hackers, are able to bypass the existing mitigations: protracted physical access to the mobile device. This access facilitates data extraction in that devices can be kept charging, and mitigations such as disabling physical data ports or remote lock or wipe can be evaded.

We consider deep physical analysis of hardware (particularly de-soldering and "de-capping"[2] the silicon, often done with nitric acid to gain direct access to underlying physical logic implementations) out of scope, as they seem to be prohibitively expensive and risk destroying device hardware or invalidating evidence; we see no clear evidence of this occurring at scale even in federal law enforcement offices. However, we do see evidence of some physical analysis in the form of academic [36] and commercial [37] research, to the extent of interposing the device's connections to storage or power.

In some cases, law enforcement receive consent from the targets of investigation to access mobile devices. This consent is likely accompanied with passcodes, PINs, and/or passwords. A database of recent warrants against iOS devices shows that not only do law enforcement agents sometimes get this consent, they also seek warrants which nullify any later withdrawal of consent [38] and as such can use their position and access to completely compromise the device.

In other cases, law enforcement agencies are able to execute warrants which create a "geofence" or physical region and period of time, such that any devices which are found to have been present therein are subject to search, usually in the form of requesting data from

[1] Examples include the TrustZone architecture included in many ARM processors [35].

[2] Removing the top physical layer of shielding which covers the chip.

cloud providers (Apple for iOS devices, and Google for Android) [39]. Such geofence warrants have massive reach and potential to violate the privacy of innocent passerby, but their use is a matter of policy and thus not in scope for our analysis.

Largely, the evidence we gather and present demonstrates that this physical access is used by law enforcement to attach commercial forensic tools which perform exploitation and data extraction on mobile devices. This approach can provide both data directly from the device, or cloud access tokens resident in device memory which can be further exploited to gain access to a target's online accounts and content. These two methods, device and cloud extraction, can provide overlapping but different categories of sensitive personal data, and, together or individually, represent a massive breach of a target's privacy.

2.3 Sensitive Data on Phones

The U.S. National Institute of Standards and Technology (NIST) maintains a list of data targets for mobile device acquisition forensic software. These are presented in Figure 2.1. The categories of data which forensic software tests attempt to extract provide us with a notion of what data is prioritized by law enforcement, and allow us to focus our examination of user data protection. The importance of these categories is corroborated by over 500 warrants against iPhones recently collected and released in the news [38, 40], and articles posted by mobile forensics companies and investigators [6, 41].

While a useful resource, this list does not capture the extent of potential privacy loss due to unauthorized mobile device access, primarily falling short in two ways. First, it does not capture the long-lived nature of some identifiers, nor the potential sensitivity of each item. Second, critically, mobile devices contain information about and from ourselves but also our networks of peers, friends, and family, and so privacy loss propagates accordingly. Further, due to the emerging capabilities of machine learning and data science techniques combined with continuously increasing availability of aggregated data sets, predictions and analysis (whether correct or not) make these potential violations of privacy nearly unbounded.

Figure 2.1: List of Targets for NIST Mobile Device Acquisition Forensics

- Cellular network subscriber information: IMEI, MEID/ESN
- Personal Information Management (PIM) data: address book/contacts, calendar, memos, etc
- Call logs: incoming, outgoing, missed
- Text messages: SMS, MMS (audio, graphic, video)
- Instant messages
- Stand-alone files:[a] audio, documents, graphic, video
- E-mail
- Web activity: history, bookmarks
- GPS and geo-location data
- Social media data: accounts, content
- SIM/UICC data: provider, IMSI, MSISDN, etc

In accordance with NIST standards, DHS tests forensic software for mobile device acquisition of these categories of data [22, 42].

Source: NIST [42]

[a] This list has evolved over time, and certain entries (such as "stand-alone files") seem to imply a disconnect between this list and how modern smartphones organize and store personal information, and may be remnants of how multimedia/feature phones used to do so. Thus is is unclear if third-party app data falls into this category.

Chapter 3

Apple iOS

Apple devices are ubiquitous in countries around the world. In Q4 of 2019 alone, almost 73 million iPhones and almost 16 million iPads shipped [43, 44]. While Apple devices represent a minority of the global smartphone market share, Apple maintains approximately a 48% share of the smartphone market in the United States [45], with similar percentages in many western nations. Overall, Apple claims 1.4 billion active devices in the world [46]. Along with increasing usage trends, these factors make iPhones extremely valuable targets for hackers, with bug bounty programs offering up to $2 million USD [47, 48], for law enforcement agencies executing warrants, and for governments seeking to surveil journalists, activists, or criminals [49].

Apple invests heavily in restricting the operating system and application software that can run on their hardware [27, 28]. As such, even users with technical expertise are limited in their ability to extend and protect Apple devices with their own modifications, and Apple software development teams represent essentially the sole *technical* mitigation against novel attempts to access user data without authorization. The high value of Apple software exploits and Apple's centralized response produces a cat-and-mouse game of exploitation and patching, where users can never be fully assured that their device is not vulnerable. Apple undertakes protecting user devices through numerous and varied mitigation strategies, and while these include both technical and business approaches, the technical will be primarily and thoroughly examined in this work.

3.1 Protection of User Data in iOS

In this section we provide an overview of key elements of Apple's user data protection strategy that cover the bulk of on-device and cloud interactions supported by iOS devices. This overview is largely based on information published by Apple [27, 28], and additionally on external (to Apple) research, product analyses, and security tests.

User authentication. Physical interaction is the primary medium of modern smartphones. In order to secure a device against unauthorized physical access, some form of user authentication is needed. iOS devices provide two mechanisms for this: (1) numeric or alphanumeric

passcodes and (2) biometric authentication. In early iPhones, Apple suggested a default of four-digit numeric passwords, but now suggests a six-digit numeric passcode. Users may additionally opt for longer alphanumeric passphrases, or (against Apple's advice [28]) disable passcode authentication entirely.

Because there are a relatively small number of 6-digit passcodes, iOS is designed to rate-limit passcode entry attempts in order to prevent attackers from conducting brute-force guessing attacks. In the event of an excessive number of entry attempts, device access may be temporarily locked and user data can be permanently deleted. To improve the user experience while maintaining security, Apple employs biometric access techniques in its devices: these include TouchID, based on a capacitive fingerprint sensor, and a more recent replacement FaceID, which employs face recognition using a depth-sensitive camera [27, 28]. The image in Figure 3.1 demonstrates the TouchID and six-digit passcode setup interfaces on iOS.

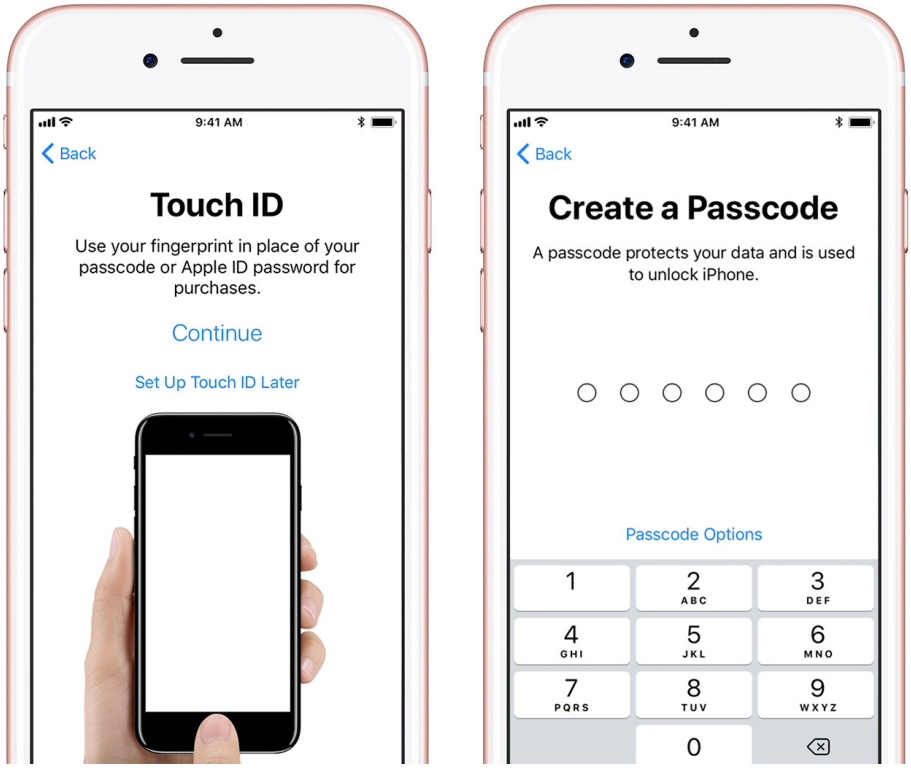

Figure 3.1: iPhone Passcode Setup Interface

Source: Apple [50]

Apple also restricts a number of passcodes that are deemed too common, or too "easily guessed." For a thorough examination of this list and its effects on iOS, refer to recent works by Markert et al. [51].

Code signing. iOS tightly restricts the executable code that can be run on the platform. This is enforced using digital signatures [27, 28]. The mechanics of iOS require that only

software written by Apple or by an approved developer can be executed on the device.

Trusted boot. Apple implements signatures for the software which initializes the operating system, and the operating system itself, in order to verify its integrity [27, 28]. These signature checks are embedded in the low-level firmware called Boot ROM. The primary purpose of this security measure, according to Apple, is to ensure that long-term encryption keys are protected and that no malicious software runs on the device.

App signing. Apple authorizes developers to distribute code using a combination of Apple-controlled signatures and a public-key certificate infrastructure that allows the system to scale [28, 52]. Organizations may also apply and pay for *enterprise signing certificates* that allow them to authenticate software for specially-authorized iOS devices (those that have installed the organization's certificate [28, 53]). This is intended to enable companies to deliver proprietary internal apps to employees, although the mechanism has been subverted many times for jailbreaking [54], for advertising or copyright fraud [55], and for device compromise [56]. The image in Figure 3.2 displays Apple documentation of the code signing process for developers.

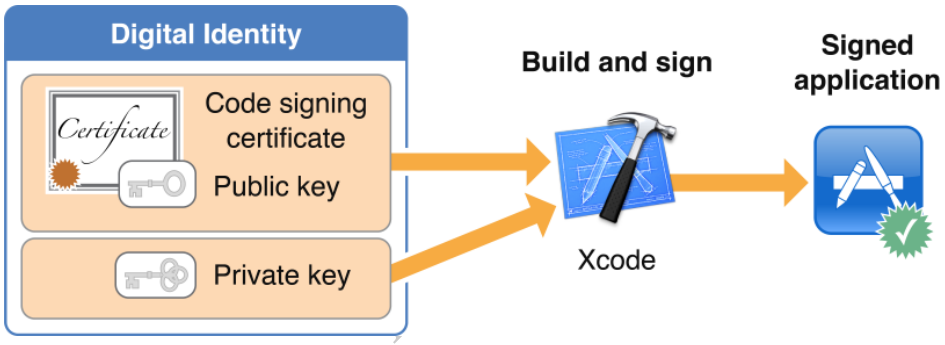

Figure 3.2: Apple Code Signing Process Documentation

Source: Apple Developer Documentation [57]

Sandboxing and code review. For third-party code, iOS enforces restrictions that limit each application's access to user data and the operating system APIs. This mechanism is designed to protect against incursions by malicious third-party applications, and to limit the damage caused by exploitation of a non-malicious application. To implement this, iOS runs each application in a "sandbox" that restricts its access to the device filesystem and memory space, and carries a signed manifest that details allowed access to system services such as location services. For applications distributed via its App Store – which is the only software installation method allowed on a device with default settings – Apple additionally performs automated and manual code review of third-party applications [58]. Despite these protections, malicious or privacy-violating applications have passed review [59].

iOS 14[1] includes additional privacy transparency and control features such as listing

[1]Released contemporaneously with this writing.

privacy-relevant permissions in the App Store, allowing finer-grained access to photos, an OS-supported recording indicator, Wi-Fi network identifier obfuscation, and optional approximate location services [60]. However, most of these features are focused on the privacy of users from app developers rather than from the phone itself, the relevant adversary under the threat model of forensics.

Encryption. While software protections provide a degree of security for system and application data, these security mechanisms can be bypassed by exploiting logic vulnerabilities in software or flaws in device hardware. Apple attempts to address this concern through the use of data encryption. This approach provides Apple devices with two major benefits: first, it ensures that Apple device storage can be rapidly erased, simply by destroying a single encryption key carried within the device. Second, encryption allows Apple to provide strong file-based access control to files and data objects on the device, even in the event that an attacker bypasses security controls within the operating system. The image in Figure 3.3 depicts the key hierarchy used in iOS Data Protection.

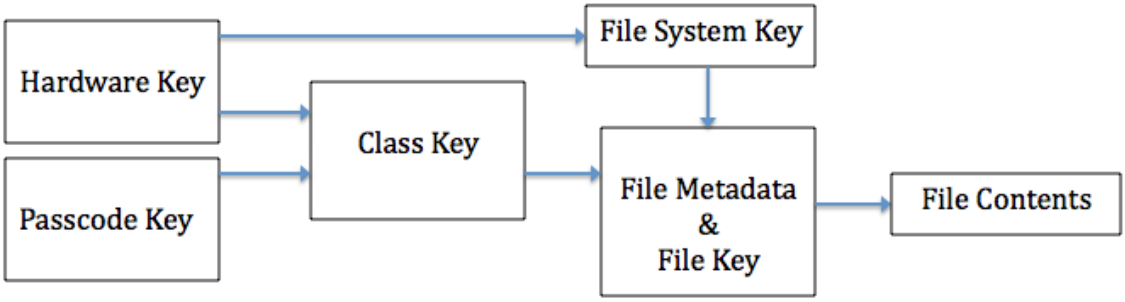

Figure 3.3: iOS Data Protection Key Hierarchy. Each arrow

Source: Washington University in St. Louis [61]

iOS employs industry-standard cryptography including AES [62], ECDH over Curve25519 [63], and various NIST-approved standard constructions to encrypt files in the filesystem [27, 28]. To ensure that data access is controlled by the user and is tied to a specific device, Apple encrypts all files using a key that is derived from a combination of the user-selected passcode and a unique secret cryptographic key (called the UID key) that is stored within device hardware. In order to recover the necessary decryption keys following a reboot the user must enter the device passcode. When the device is locked but has been unlocked since last boot, biometrics suffice to unlock these keys. To prevent the user from bypassing the encryption by guessing a large number of passcodes, the system enforces guessing limits in two ways: (1) by using a computationally-intensive password-based key derivation function that requires 80ms to derive a key on the device hardware [27, 28], and (2) by enforcing guess limits and increasing time intervals using trusted hardware (see further below) and software [27, 28].

Data Protection Classes. Apple provides interfaces to enable encryption in both first-party and third-party software, using the iOS Data Protection API [27, 28]. Within this package, Apple specifies several encryption "protection classes" that application developers can select

when creating new data files and objects. These classes allow developers to specify the security properties of each piece of encrypted data, including whether the keys corresponding to that data will be evicted from memory after the phone is locked ("Complete Protection" or CP) or shut down ("After First Unlock" or AFU).

We present a complete list of Data Protection classes in Figure 3.4. As we will discuss below *the selection of protection class makes an enormous practical difference in the security afforded by Apple's file encryption.* Since in practice, users reboot their phones only rarely, many phones are routinely carried in a locked-but-authenticated state (AFU). This means that for protection classes other than CP, decryption keys remain available in the device's memory. Analysis of forensic tools shows that to an attacker who obtains a phone in this state, encryption provides only a modest additional protection over the software security and authentication measures described above.

Figure 3.4: List of iOS Data Protection Classes

Complete Protection (CP): Encryption keys for this data are evicted shortly after device lock (10 seconds).

Protected Unless Open (PUO): Using public-key encryption, PUO allows data files to be created and encrypted while the device is locked, but only decrypted when the device is unlocked, by keeping an ephemeral public key in memory but evicting the private key at device lock. Once the file has been created and closed, data in this class has properties similar to Complete Protection.

Protected Until First User Authentication (a.k.a. **After First Unlock**) **(AFU)**: Encryption keys are decrypted into memory when the user first enters the device passcode, and remain in memory even if the device is locked.

No Protection (NP): Encryption keys are encrypted by the hardware UID keys only, not the user passcode, when the device is off. These keys are always available in memory when the device is on.

Source: Apple iOS and Platform Security Guides [27, 28]

A natural question arises: why not simply apply CP to all classes of data? This would seriously hamper unauthorized attempts to access user data. However, the answer appears to lie in user experience. Examining the data in Table 3.10 it seems likely that data which is useful to apps which run in the background, including while the device is locked, is kept in the AFU state in order to enable continuous services such as VPNs, email synchronization, contacts, and iMessage. For example, if user's contacts were protected using CP, then a locked phone would be unable to display a name associated with a phone number for an

incoming text message, and likely would just display the phone number itself.[2] This would severely impact the feature of iOS to preview sender and message content in lock screen notifications.

Keychain. iOS provides the system and applications with a secure key-value store API called the Keychain [27, 28] for storing sensitive secrets such as keys and passwords. The Keychain provides encrypted storage and permissioned access to these secret items via a public API which third-party app developers can build around. This Keychain data is encrypted using keys that are in turn protected by device hardware keys and the user passcode, and can optionally placed into protection classes that mirror the protection classes in Figure 3.4. In addition to these protection classes, the Keychain also introduces an optional characteristic called Non-Migratory (NM), which ensures that any protected data can only be decrypted on the same device that it was encrypted under. This mechanism is enforced via the internal hardware UID key, which cannot be transported off of the device.[3]

Apple selects the Data Protection classes used by built-in applications such as iMessage and Photos, while third-party developers may choose these classes when developing applications. If they do not explicitly select a different protection class, the default class used is Protected Until First User Authentication, or AFU. Their documentation [27, 28] claims that, at least, the following applications' data falls under some degree of Data Protection: Messages, Mail, Calendar, Contacts, Photos, and Health, in addition to all third-party apps in iOS 7 and later. Refer to Table 3.10 for more details on Data Protection, and to Figure 3.17 for details on data which is necessarily AFU or less protected due to its availability via forensic tools.

Backups. iOS devices can be backed up either to iCloud or to a personal computer. When backing up to a personal computer, users may set an optional backup password. This password is used to derive an encryption key that, in turn, protects a structure called the "Keybag." The Keybag contains a bundle of further encryption keys that encrypt the backup in its entirety. A limitation of this mechanism is that the user-specified backup password must be extremely strong: since this password is not entangled with device hardware secrets, stored backups may be subject to sophisticated offline dictionary attacks that can guess weak passwords, however iOS uses 10 million iterations of PBKDF2 [64] to significantly inhibit such password cracking [27, 28].[4] iOS devices may also be configured to backup to Apple iCloud servers. In this instance, data is encrypted asymmetrically using Curve25519 [63] (so that backups can be performed when the device is locked without exposing secret keys) [27], and those keys are encrypted with "iCloud keys" known to Apple to create the iCloud Backup Keybag. This means that Apple itself[5] or a malicious actor who can guess or reset user credentials can access the contents of the backup. For both types of backups, the Keychain is additionally encrypted with a key derived from the hardware UID key to prevent restoring it to a new device [27, 28].

[2]This is indeed the behavior that phones exhibit prior to the first unlock.

[3]Use of this class ensures that protected data cannot be restored onto a new phone and decrypted.

[4]Since iOS 10.2, previously 10,000; compare iOS security guides for iOS 9 (May 2016) and iOS 10 (March 2017) [27].

[5]Acting on its own or under court order.

Aside from Mail, which is not encrypted at rest on the server [31], all other backup data is stored encrypted with keys that Apple has access to. This implies that such data can be accessed through an unauthorized compromise of Apple's network, a stolen credential attack, or compelled access by authorized government officials. The data classes included in an iCloud backup are listed in Figure 3.5.

Figure 3.5: List of Data Included in iCloud Backup

- App data[a]
- Apple Watch backups
- Device settings
- Home screen and app organization
- iMessage, SMS, and MMS messages[b]
- Photos and videos
- Purchase history from Apple services
- Ringtones
- Visual Voicemail password

Source: Apple Support Documentation [65]

[a]Unless developers opt out, refer to §3.1.
[b]iMessages are stored in iCloud, rather than in backups, if iCloud for iMessage is enabled.

iCloud. In addition to backups, iCloud can be used to store and synchronize various classes of data, primarily for built-in default apps such as Photos and Documents. Third-party apps' files are also included in iCloud Backups unless the developers specifically opt-out [66]. Apple encrypts data in transit using Transport Layer Security (TLS) as is standard for internet traffic [31]. Data at rest, however, is a more complex story: Mail is stored *unencrypted on the server* (which Apple claims is an industry standard practice [31]). The data classes in Figure 3.6 is stored encrypted with a 128-bit AES key known to Apple, and the data classes in Figure 3.7 is stored encrypted with a key derived from the user passcode[6] and is thus protected from even Apple. There are caveats to these lists, including that Health data is only end-to-end encrypted if two-factor authentication is enabled for iCloud [31], and that Messages in iCloud, which can be enabled in the iOS settings, uses end-to-end encryption, but the key is also included in iCloud backups and thus can be accessed by Apple if iCloud backup is enabled [31].

The user experience of controlling access to iCloud data embeds relatively unpredictable aspects: for example, disabling iCloud for the default Calendar app prevents the sending of calendar invites via email on iOS 13.[7] A variety of exceptions and special cases, like the iMessage example above, combined with unpredictable side-effects on user experience, makes it more difficult for users to secure a device by adjusting user-facing settings.

[6]Potentially also the hardware UID key, although Apple claims that this data can be recovered on a new device using only the passcode [31].
[7]This was confirmed experimentally by the authors.

Figure 3.6: List of iCloud Data Accessible by Apple

- Safari History & Bookmarks[a]
- Calendars
- Contacts
- Find My[b]
- iCloud Drive[c]
- Messages in iCloud
- Notes
- Photos
- Reminders
- Siri Shortcuts
- Voice Memos
- Wallet Passes[d]

Source: Apple iCloud Security Guide [31]

[a]History is stored end-to-end encrypted on iOS 13 and later.
[b]List of devices and people enrolled in Find My, an Apple service for physically locating enrolled devices [67].
[c]Used for document storage for Apple's office suite (Pages, Keynote, and Numbers).
[d]Items such as ID cards, boarding passes for airlines, and other supported content.

Figure 3.7: List of iCloud Data Encrypted "End-to-End"

- Apple Card transactions
- Home data
- Health data
- iCloud Keychain[a]
- Maps data[b]
- Memoji[c]
- Payment information
- Quicktype[d] Keyboard learned vocabulary
- Safari History and iCloud Tabs
- Screen Time
- Siri information[e]
- Wi-Fi passwords
- W1 and H1[f] Bluetooth keys

Source: Apple iCloud Security Guide [31]

[a]See "iCloud Keychain" in this section.
[b]Favorite locations and other location history including searches.
[c]Generated emoji-style images.
[d]Word suggestions above the keyboard on iOS, introduced in iOS 8 [68].
[e]It is not clear what specific data this contains, the documentation is vague.
[f]Apple product names for wireless-enabled integrated circuits.

CloudKit. Third-party developers can also integrate with iCloud in iOS applications via CloudKit, an API which allows applications to access cloud storage with configurable properties that allow data to be shared in real-time across multiple devices [52]. CloudKit data

is encrypted into one or more "containers" under a key hierarchy similar to Data Protection. The top-level key in this hierarchy is the "CloudKit Service Key." This key is stored in the synchronized user Keychain, inaccessible to Apple, and is rotated any time the user disables iCloud Backup [28, 31, 69].

iCloud Keychain. iCloud Keychain extends iCloud functionality to provide two services for Apple devices: (1) Keychain synchronization across devices, and (2) Keychain recovery in case of device loss or failure [28].

1. Keychain syncing enables trusted devices (see below) to share Keychain data with one another using asymmetric encryption [28]. The data available for Keychain syncing includes Safari user data (usernames, passwords, and credit card numbers), Wi-Fi passwords, and HomeKit[8] encryption keys. Third-party applications may opt-in to have their Keychain data synchronized.

2. Keychain recovery allows a user to escrow an encrypted copy of their Keychain with Apple [28]. The Keychain is encrypted with a "strong passcode" known as the iCloud Security code, discussed below.

Apple documentation [70] defines a "trusted device" as:

> "an iPhone, iPad, or iPod touch with iOS 9 and later, or Mac with OS X El Capitan and later that you've already signed in to using two-factor authentication. It's a device we know is yours and that can be used to verify your identity by displaying a verification code from Apple when you sign in on a different device or browser."

In contrast with standard iCloud and iCloud backups, iCloud Keychain provides additional security guarantees for stored data. This is enforced through the use of trusted Hardware Security Modules (HSMs) within Apple's back-end data centers. This system is designed by Apple to ensure that even Apple itself cannot access the contents of iCloud Keychain backups without access to the iCloud Security Code. This code is generated either from the user's passcode if Two-Factor Authentication (2FA) is enabled, or chosen by the user (optionally generated on-device) if 2FA is not enabled. When authenticating to the Hardware Security Modules (HSMs) which protect iCloud Keychain recovery, the iCloud Security Code is never transmitted, instead using the Secure Remote Password (SRP) protocol [71]. After the HSM cluster verifies that 10 failed attempts have not occurred, it sends a copy of the encrypted Keychain to the device for decryption. If 10 attempts is exceeded, the record is destroyed and the user must re-enroll in iCloud Keychain to continue using its features.[9] As a final note, the software installed on Apple HSMs can run must be digitally signed; Apple asserts that the signing keys required to further alter the software are physically destroyed after HSM deployment [28, 72], preventing the company from deliberately modifying these systems to access user data.

[8] Used for control of smart home devices.

[9] Re-enrolling appears to be a matter of toggling the selector from off to on in Settings on an iOS device.

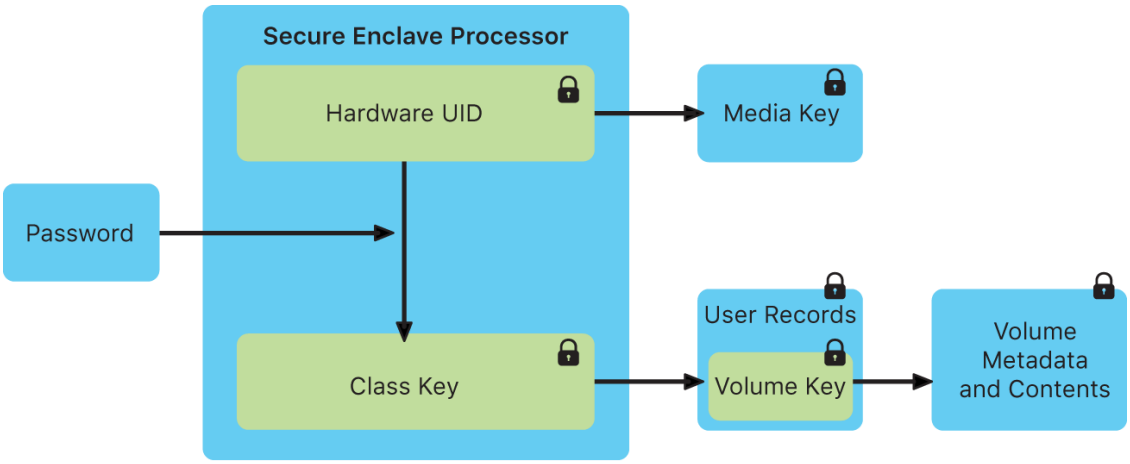

Figure 3.8: Secure Enclave Processor Key Derivation

Source: Apple [28]

Trusted hardware. Apple provides multiple dedicated components to support encryption, biometrics and other security functions. Primary among these is the Secure Enclave Processor (SEP), a dedicated co-processor which uses encrypted memory and handles processes related to user authentication, secret key management, encryption, and random number generation. The SEP is intended to enable security against even a compromised iOS kernel by executing cryptographic operations in separate dedicated hardware. The SEP negotiates keys to communicate with TouchID and FaceID subsystems (the touch sensor or the Neural Engine for facial recognition) [73], provides keys to the Crypto Engine (a cryptography accelerator), and communicates with a "secure storage integrated circuit" for random number generation, anti-replay counters, and tamper resistance [74].[10] The image in Figure 3.8 documents the encryption keys the SEP derives from the user passcode.

Apple documentation describes FaceID as leveraging the Neural Engine and the SEP: "A portion of the A11 Bionic processor's neural engine—protected within the Secure Enclave—transforms this data into a mathematical representation and compares that representation to the enrolled facial data." [73] As worded, this description fails to fully convey the features of hardware which make this possible, and as such we must speculate as to the exact method by which FaceID data is processed and secured. For example, it is possible that the Neural Engine simply provides data directly to the SEP, or via the application processor (AP). It's also possible that, similar to TouchID, the Neural Engine creates a shared key with the SEP and then passes encrypted data through the AP. Physical inspection of the iPhone X (the first generation with FFaceID) suggests that the Neural Engine and SEP are inside the A11 package [75], as in the earlier design with the SEP and A7 SoC in iPhone 5s [76].

[10]The "SEPOS" root task of the SEP, as well as various drivers, services, and applications which run in the SEP were presented in-depth at Black Hat 2016 [74]; in this work the authors elucidate the inner workings, software security mitigations implemented (with some notably absent, including ASLR and Heap metadata protection), and attack surface of the SEP as implemented in iPhone 6S/iOS 9.

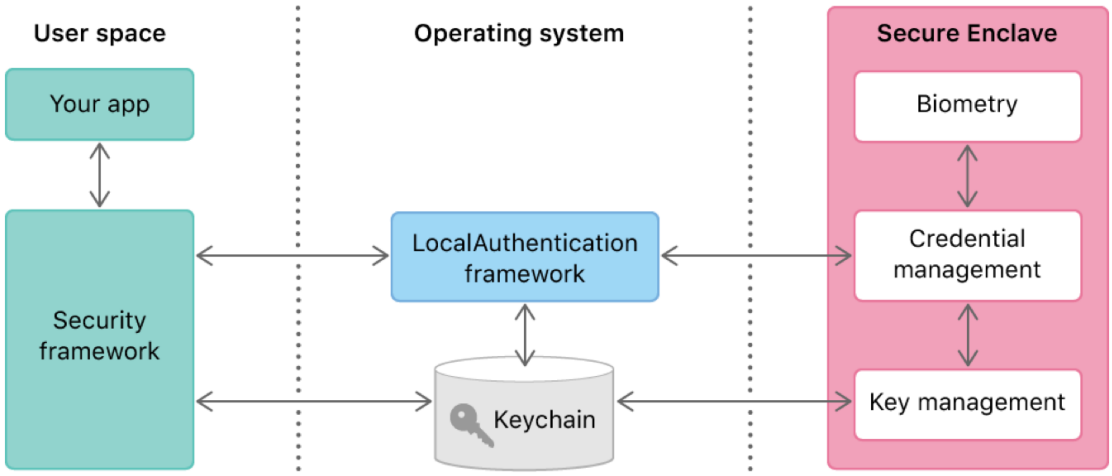

Figure 3.9: `LocalAuthentication` Interface for TouchID/FaceID

Source: Apple [78]

iPhones 6 and later (among other contemporaneous devices e.g. Apple Watches and some iPads) additionally support Apple Pay and other NFC/Suica[11] communication of secrets via the Secure Element, a wireless-enabled chip which stores encrypted payment card and other data [77]. The image in Figure 3.9 documents the `LocalAuthentication` framework through which TouchID and FaceID operate.

Restricting attack surface. A locked iOS device has very few methods of data ingress available: background downloads such as sync (email, cloud services, etc), Push notifications, certain network messages via Wi-Fi, Bluetooth, or the cellular network, and the physical Lightning port on the device. Exploits delivered to a locked device must necessarily use one of these avenues. With the introduction of USB Restricted Mode, Apple sought to limit the access of untrusted USB Lightning devices to access or interact with iOS, such as forensic software [79]. By reducing attack surface, iOS complicates or mitigates attacks. This protection mode simply disables USB communication for unknown devices after the iOS device is locked for an hour without USB accessory activity, or upon first use of a USB accessory; known devices are remembered for 30 days [80]. If no accessories are used for three days, the lockout period is reduced from an hour to immediately after device lock [80]. However, this protection is not complete, as discussed in §3.3.

iMessage and FaceTime. Each iPhone ships with two integrated communication packages: iMessage and FaceTime. The iMessage platform is incorporated within the Messages application as an alternative to SMS/MMS, and provides end-to-end encrypted messaging communications with other Apple devices. The FaceTime app provides end-to-end encrypted audio and videoconferencing between Apple devices. iMessage messages are encrypted using a "signcryption" scheme recently analyzed by Bellare and Stepanovs [81] with the Apple

[11] A Japanese contactless/wireless protocol and rough equivalent to NFC

Identity Service serving as a trusted authority for user public keys [28]. FaceTime calls are encrypted using a scheme that Apple has not documented, but claims is forward-secure[12] based on the Secure Real-time Transport Protocol (SRTP) [82]. Although these protocols are end-to-end encrypted and authenticated, they rely on the Apple Identity Service to ensure that participants are authentic [28].

3.2 History of Apple Security Features

The current hardware and software security capabilities of iOS devices are many and varied, ranging from access control to cryptography. These features were incrementally developed over time and delivered in new versions of iOS pre-loaded on or delivered to[13] devices. An overwhelming percentage of iOS users update their devices: in June 2020, Apple found that over 92% of recent[14] iPhones ran iOS 13, and almost all the remainder (7%) ran iOS 12 [83]; iOS 13 was released 9 months prior. The implication of this is that while most users receive the latest mitigations, users of older devices may receive only partial security against known attacks, especially when using devices which have reached end-of-life (no longer receiving updates from Apple). To make these limitations apparent, in Appendix A we provide a detailed overview of the historical deployment of new security features, as described by published Apple documents [27, 28, 31, 69, 70, 73, 77, 84, 85]. This history is summarized in Tables 3.10 and 3.11.

[12]Previous call content cannot be accessed even if a device or key is compromised.
[13]First via iTunes, and then over-the-air via Wi-Fi since iOS 5 in 2011.
[14]Sold new in the last four years.

iOS[a]	**Data Protection**[b]	**Notes**
1 (2G)	-	4-digit passcodes
2 (3G)	-	Option to erase user data after 10 failed passcode attempts introduced
3 (3GS)	DP introduced	Encrypted flash storage when device off
4 (4)	Mail & Attachments: PUO	PUO insecure due to implementation error on iPhone 4 until iOS 7 [84]
5 (4S)	-	AES-GCM replaces CBC in Keychain
6 (5)	iTunes Backup: CP+NM Location Data: CP Mail Accounts: AFU iMessage keys: NP+NM	Various other data at AFU or NP+NM class
7 (5S)	Safari Passwords: CP[c] Authentication tokens:[d] AFU Default and third-party apps: AFU	SEP and TouchID introduced in iPhone 5S Third-party apps may opt-in to higher classes
8 (6)	App Documents: CP Location Data: AFU?[e]	User passcode mixed into encryption keys XTS replaces CBC for storage encryption
9 (6S)	Safari Bookmarks: CP	6-digit passcode default introduced
10 (7)	Clipboard: ?[f] iCloud private key: NP+NM	-
11 (8 & X)	-	FaceID introduced in iPhone X SEP memory adds an "integrity tree" to prevent replay attacks USB Restricted Mode introduced to mitigate exploits delivered over Lightning connector
12 (XS)	-	SEP enforces Data Protection in DFU (Device Firmware Upgrade) and Recovery mode to mitigate bypass via bootloader
13 (11)	-	-

Table 3.10: History of iOS Data Protection

Source: Apple iOS and Platform Security Guides [27, 28]

[a](Model), alternate models omitted.
[b]See Figure 3.4.
[c]Possibly earlier, documentation ambiguous.
[d]Social media accounts and iCloud.
[e]Documentation includes notably weakened language.
[f]Documentation ambiguous.

SoC[ab]	**Hardware Changes**	**Notes**
Samsung ARM-32 CPU (2G)	-	No dedicated hardware security components in the early phones
Samsung ARM-32 CPU (3G)	-	-
Samsung Cortex SoC (3GS)	-	Flash storage encryption driven by application processor (AP)
Apple A4 (4)	-	Shift to Apple-designed SoCs
A5 (4S)	Crypto Engine	Dedicated cryptography accelerator documented here, potentially included in earlier generations
A6 (5)	-	-
A7 (5S)	TouchID and Secure Enclave Processor (SEP)	Major change, including new UID key inside SEP and shared key between SEP and TouchID sensor
A8 (6)	-	Significant software changes in this generation which rely on hardware changes of A7
A9 (6S)	Bus between flash and memory "isolated via the Crypto Engine"	Interpreting this documentation implies that hardware changed to physically enforce flash storage encryption
A10 Fusion (7)	-	-
A11 Bionic (8/X)	FaceID and Neural Engine	Neural Engine somehow integrated with SEP to enable facial recognition secure against malicious AP
A12 Bionic (XS)	"Secure Storage Integrated Circuit"	SSIC added to bolster SEP replay protection, RNG, and tamper detection
A13 Bionic (11)	-	-

Table 3.11: History of iPhone Hardware Security

Source: Apple iOS and Platform Security Guides [27, 28] and iFixit Teardowns [75, 76, 86, 87, 88, 89, 90, 91, 92, 93]

[a] System-on-a-Chip.
[b] (Model), alternate models omitted.

3.3 Known Data Security Bypass Techniques

As each iteration of Apple device introduces or improves on security features, the commercial exploit/forensics and jailbreaking communities have reacted by developing new techniques to bypass those features. Bypassing iPhone protections is an attractive goal for threat actors and law enforcement agencies alike. Hackers can receive bug bounties amounting to six or seven digits for viable exploits [47, 48]; rogue governments can buy or develop and use malware to track human rights activists or political opponents [49]; and law enforcement agencies can create or bolster cases through investigation of phone contents, with forensic software companies signing lucrative [94] contracts to help them do so [6, 7].

Law enforcement agencies in many jurisdictions may also provide legal requests for data. In pursuing compliance with these laws, Apple provides data to law enforcement when legal requests are made. Figure 3.12 documents data which Apple claims to be able or not able to provide to U.S. law enforcement [95]. This method of access requires legal process, and certain information being provided to Apple, and some requests may be rejected under various circumstances [26]. As such, this method of data collection may be supplemented or even entirely superseded by commercial data forensics methods described in §3.3.2 and §3.3.3.

Jailbreaks and software exploits. The primary means to bypass iOS security features is through the exploitation of software vulnerabilities in apps and iOS software. Jailbreaks are a class of such exploits central to forensic analysis of iOS. The defining feature of a jailbreak is to enable running unsigned code, such as a modified iOS kernel or apps that have not been approved by Apple [96] or a trusted developer [53]. Other software exploits which pertain to user data privacy (speculatively) include SEP exploits, exploits which may enable passcode brute-force guessing and lock-screen bypasses. All of these exploits are discussed in §3.3.1.

Local device forensic data extraction. Once a device has been made accessible using a software exploit, actors may require technological assistance to perform forensic analysis of the resulting data. This requires tools that extract data from a device and render the results in a human-readable form [42, 22]. This latter niche has been actively filled by a variety of private companies, whose software is tested and used by the U.S. Department of Homeland Security (DHS) and local law enforcement, with public reporting on its effectiveness [22] (refer to Figure 2.1 for the data targeted by mobile forensic software). DHS evaluations of forensic software tools reveal that, at least in laboratory settings, significant portions of the targeted data is successfully extracted from supported devices. Refer to §3.3.2 for discussion of the use of such tools, and to §3.4 for further examination of the history of forensic software.

Cloud forensic data extraction. Cloud integrations such as Apple iCloud enable valuable features such as backup and sync. They also create a data-rich pathway for information extraction, and represent a target for search warrants themselves. The various ways these services can be leveraged by hackers or law enforcement forensics are discussed in §3.3.3.

Figure 3.12: Apple Documentation on Legal Requests for Data

Data Apple makes available to law enforcement:

- Product registration information[a]
- Customer service records
- iTunes subscriber information
- iTunes connection logs (IP address)
- iTunes purchase and download history
- Apple online store purchase information[b]
- iCloud subscriber information
- iCloud email logs (metadata)
- iCloud email content
- iCloud photos
- iCloud Drive documents
- Contacts
- Calendars
- Bookmarks
- Safari browsing history
- Maps search history
- Messages (SMS/MMS)
- Backups (app data, settings, and other data)
- Find My [67] connections and transactions[c]
- Hardware address (MAC) for a given iPhone[d]
- FaceTime call invitations[e]
- iMessage capability query logs[f]

Data Apple claims is unavailable:

- Find My location data
- Full data extractions/user passcodes on iPhone 6/iOS 8.0 and later
- FaceTime call content
- iMessage content

Source: Apple Legal [95]

[a]Including name, address, email address, and telephone number.
[b]Linked to iCloud account.
[c]Remote lock or erase requests
[d]By serial number or mobile subscriber information (IMEI, MEID, UDID).
[e]With the caveat that these do not imply any communication took place.
[f]When an iOS device encounters a potential iMessage handle (phone number, email address, or Apple ID), it queries Apple to determine if the contact is able to use iMessage.

3.3.1 Jailbreaking and Software Exploits

Jailbreaking. While jailbreaks are often used for customization purposes, the underlying technology can also be used by third parties in order to bypass software protection features of a target device, for example to bypass passcode lock screens or to forensically extract data from a locked device. Indeed, many commercial forensics packages make use of public jailbreaks released for this purpose [97]. A commonality among these techniques is the need

to deliver an "exploit payload" to a vulnerable software component on the device. Viable delivery mechanisms include physical ports such as the device's USB/Lightning interface (much less viable if USB Restricted Mode is active [85]). Alternatively, exploits may be delivered through data messages that are received and processed by iOS and app software, for example specially-crafted text messages or web pages. In many cases, multiple separate exploits are combined to form an "exploit chain:" the first exploit may obtain control of one software system on the device, while further exploits may escalate control until the kernel has been breached. Once the kernel has been exploited, jailbreaks usually deploy a patch to allow unsigned code to run or to initiate custom behavior such as extraction of the filesystem [97].

Jailbreaking is a keystone of constructing bypasses to access user data due to the fact that the iOS kernel is ultimately tasked with managing and retrieving sensitive data. As such, a kernel compromise often allows the immediate extraction of any data not explicitly protected by encryption using keys which cannot be derived from the application processor alone. Publicly-known jailbreaks are released by a seemingly small group of exploit developers.[15] Jailbreaks are released targeting a specific iOS version, and more rarely target specific hardware (e.g. iPhone model). Apple periodically releases software updates which patch some subset of the vulnerabilities distributed in these jailbreaks [84], and a process ensues in which exploit developers replace patched vulnerabilities with newly discovered, still-exploitable alternatives, until a major software change occurs (e.g. a new iOS major version). Table 3.15 provides the highlights of the history of jailbreaking in iOS, with many of these iterative updates omitted.

Jailbreaking was relatively popular in 2009. Exact counting is nearly impossible, but it was estimated that 10% of iOS devices in 2009 were jailbroken [98].[16] In 2013, roughly 23 million devices ran Cydia [96], a popular software platform commonly used on jailbroken iOS devices [98]. 150 million iPhones were sold that year [99] and total iPhone sales accelerated tremendously between 2009 and 2013 [100], and as such it is speculatively likely, though hard to measure, that the percentage of jailbroken devices declined notably. Some analysis has been undertaken as to reasons for declining jailbreaking, if this even is a trend [98]. Around 2016 (refer to Table 3.15) there was a marked transition in jailbreaks away from end-user usability and towards support for use by security researchers. Some of the more well-maintained jailbreaks did include single-click functionality or re-jailbreaking after reboot via a sideloaded[17] app. It is possible that this transition took place in part due to the commercial viability of jailbreak production. As the market is relatively inflexibly supplied, prices for working jailbreaks increase directly with demand [48], and as such creators are less inclined to share exploits which are necessary for jailbreaking publicly. The kinds of exploits needed for forensics against a locked phone, specifically those which exploit an interface on the locked phone (commonly the USB/Lightning interface and related components), would be highly valuable to a forensics software company which at a given time did not have a working exploit of their own.

[15] As seen in Table 3.15.

[16] Representing 10.5 to 12 million devices.

[17] Installed via personal computer rather than the official App Store.

The checkm8/checkra1n jailbreak exploits seem to be widely implemented in forensic analysis tools in 2020. These exploits work on iOS devices up to iPhone X (and any with A11 hardware iterations) *regardless of iOS version* and as such are widely applicable and thus useful [101]. Cellebrite Advanced Services (their bespoke law enforcement investigative service) offers Before-First-Unlock access[18] to iPhones X and earlier running up to the latest iOS [7], and as such we are relatively certain they are employing checkm8.

Although they do not refer to it as jailbreaking, the exploits used in the Cellebrite UFED Touch and 4PC products either exploit the backup system to initiate a backup despite lacking user authorization, or exploit the bootloader to run custom Cellebrite code which extracts portions of the filesystem [14]. We categorize these as equivalent due to the practical implementation and impact similarities.

Passcode guessing. To access records on devices that are not in the AFU state, or to access data that has been protected using the CP class, actors may need to recover the user's passcode in order to derive the necessary decryption keys.

There are two primary obstacles to this process: first, because keys are derived from a combination of the hardware UID key and the user's passcode, keys must be derived *on the device*, or the UID key must be physically extracted from silicon.[19] There is no public evidence that the latter strategy is economically feasible. The second obstacle is that the iPhone significantly throttles attackers' ability to conduct passcode guessing attacks on the device: this is accomplished through the use of guessing limits enforced (on more recent phones) by the dedicated SEP processor, as well as an approximately 80 millisecond password derivation time enforced by the use of a computationally-expensive key derivation function.

In older iPhones that do not include a SEP, passcode verification and guessing limits were enforce by the application processor. Various bugs [84] in this implementation have enabled attacks which exploit the passcode attempt counter to prevent it from incrementing or to reset it between attempts. With four- and six-digit passcodes, and especially with users commonly selecting certain passcodes [51], these exploits made brute-forcing the phone feasible for law enforcement. One particularly notable example of passcode brute-forcing from a technical perspective was contributed by Skorobogatov in 2016 [36]. In this work, the authors explore the technical feasibility of mirroring flash storage on an iPhone 5C to enable unlimited passcode attempts. Because the iPhonee 5C does not include a SEP with tamper-resistant NVRAM, the essence of the attack is to replace the storage carrying the retry counter in order to reset its value. The authors demonstrate that the attack is indeed feasible, and inexpensive to perform. Even earlier attacks include cutting power to the device when an incorrect passcode is entered to preempt the counter before it increments [37], although these have largely been addressed.[20]

[18] Among other less technically challenging offerings.

[19] Private communications from Apple engineers claim that the UID key cannot be obtained directly by software, and can only be used as input to the AES engine. This is claimed to be enforced through the silicon design, rather than through software, ensuring that only expensive physical extraction can obtain the raw UID value.

[20] A standard countermeasure introduced in non-SEP phones requires the device to increment the guess counter *prior* to verifying the passcode, ensuring that data must be written to storage.

For extremely strong passwords (such as random alphanumeric passcodes, although these are relatively rarely used [51]), the 80ms guessing time may render passcode guessing attacks completely infeasible regardless of whether the SEP exists and is operating. For lower-entropy passcodes such as the default 6-digit numeric PIN, the SEP-enforced guessing limits represent the primary obstacle.[21] Bypassing these limitations requires techniques for overcoming the SEP guessing limits. We provide a detailed overview of the evidence for and against the in-the-wild existence of such exploits in §3.3. Refer to Table 3.13 for estimated passcode brute-force times under various circumstances.

Passcode Length	**4 (digits)**	**6 (digits)**	**10 (digits)**	**10 (all)**[a]
Total Passcodes	$10^4 = 10,000$	$10^6 = 1,000,000$	10^{10}	3.7×10^{20}
Total Allowed[b]	9,276	997,090	-	-
80 ms/attempt	12.37 minutes	22.16 hours	∼25 years	[c]
in expectation	*6.19 minutes*	*11.08 hours*	*∼12 years*	
10 mins/attempt[d]	∼70 days	∼20 years	∼200,000 yrs.	[e]
in expectation	*∼35 days*	*∼10 years*		

Table 3.13: Passcode Brute-Force Time Estimates

A more intelligent passcode guessing strategy could succeed much more quickly.

[a] The iOS US QWERTY keyboard allows around 114 character inputs.

[b] iOS prevents users from choosing certain 4- or 6-digit passcodes considered too "easily guessed." For more information, refer to [51].

[c] Roughly 68 times the age of the universe.

[d] From an Elcomsoft article which claimed this rate using GrayKey on a before-first-unlock device [102].

[e] Inestimable.

Exploiting the SEP. The Secure Enclave Processor is a separate device that runs with unrestricted access to the memory of the phone [74] and as such a SEP exploit provides an even greater level of access than an OS jailbreak. Moreover, exploitation of the SEP is the most likely means to bypass security mechanisms such as passcode guessing limits. In order to interface with the SEP with sufficient flexibility, kernel privileges are required [74], and thus a jailbreak is likely a prerequisite for SEP exploitation. In the case that such an exploit chain could be executed, an attacker or forensic analyst might gain unfettered access to the encryption keys and functionality used to secure the device, and thus would be able to completely extract the filesystem.

In 2018, Grayshift,[22] a forensics company based in Atlanta, Georgia, advertised and sold a device they called GrayKey, which was purportedly able to unlock a locked or disabled iPhone by brute-forcing the passcode. Modern iPhones allegedly exploited in leak GrayKey

[21] Without these limits, an attacker could guess all possible six-digit PINs in a matter of hours.

[22] `https://www.grayshift.com/`

demonstration photos [23] included a SEP, meaning that for these phones, brute-force protections should have prevented such an attack. As such, we speculate that GrayKey may have embedded not only a jailbreak but also a SEP exploit in order to enable this functionality. Other comparable forensic tools seemed only able to access a subset of the data which GrayKey promised at the time [7, 22, 97] which provides additional circumstantial evidence that SEP exploits, if they existed, are rare.

A 2018 article from the company MalwareBytes [23] provided an alleged screenshot of an iPhone X (containing a SEP) running iOS 11.2.5 (latest at the time) in a before-first-unlock state (all data in the AFU and CP protection classes encrypted, with keys evicted from memory and thus unavailable to the kernel). The images indicate that the GrayKey exploit had successfully executed a guessing attack on a 6-digit passcode, with an estimated time-to-completion of approximately 2 days, 4 hours. The images also show a full filesystem image and an iTunes backup extracted from it, which should only be possible if the passcode was known [27, 28] or somehow extracted. As the time to unlock increases with passcode complexity, we presume that GrayKey is able to launch a brute-force passcode guessing attack from within the exploited iOS, necessitating a bypass of SEP features which should otherwise prevent such an attack. Figure 3.14 depicts the Graykey passcode guessing and extraction interfaces.

In August 2018, Grayshift unlocked an iPhone X with an unknown 6-digit passcode given to them by the Seattle Police Department; Grayshift was reportedly able to break the passcode in just over two weeks [17]. Further documents [30] imply extensive use of Graykey to recover passcodes against iOS devices.

Figure 3.14: Alleged Leaked Images of the GrayKey Passcode Guessing Interface

Left: passcode guessing on an iPhone X. Right: forensic extraction. Source: MalwareBytes SecurityWorld [103]

In January 2020, an FBI warrant application indicates that GrayKey was used to access a visibly locked iPhone 11 Pro Max [104, 105]. The significance of this access is twofold: (1) the iPhone 11 Pro Max, released with iOS 13, is not vulnerable to the checkm8 jailbreak exploit [101], and (2) the iPhone 11 includes a SEP. The apparent success of this extraction

indicates that Grayshift possessed an exploit capable of compromising iOS 13. There is simply not enough information in these warrants to know if this exploit simply performed a jailbreak and logical forensic extraction of non-CP encrypted data, or if it involved a complete compromise of the SEP and a successful passcode bypass.

Also in 2020, Grayshift filed with the U.S. Federal Communications Commission (FCC) ID program for certification of the GrayKey device [106, 107]. This filing initiated the public release of documents detailing the Radio Frequency (RF) hardware and capabilities of GrayKey, as well as an image of the internals. Analysis[23] of these reports and images has induced speculation that the GrayKey device first exploits iOS then gains additional access to software systems through post-exploitation tools, potentially including by interfacing with JTAG [108], a hardware debugging interface. The GrayKey hardware also includes Wi-Fi and Bluetooth capabilities which could be used for updates, data exfiltration, or exploitation of iOS devices.

[23]Including on Twitter: `https://twitter.com/parallel_beings/status/1305554386828038146?s=20`

iOS[a]	Year	Jailbreak[b]	Author(s)	Notes
1.1.1-4 (2G)	2007	JailbreakMe	iPhone Dev Team[c]	-
2.2-5 (3G-4)	2008-	QuickPwn, redsn0w	iPhone Dev Team	Supported for multiple versions, renamed redsn0w in 2009
3 (3GS)	2009	purplera1n, blackra1n [109, 110]	geohot	First "unlock," meaning the cell network carrier can be changed
5 (4S)	2012	Absinthe	Multiple	Notable collaboration of iPhone Dev Team, Chronic Dev Team, and others
6-7 (4S)	2013-	evasi0n, evasi0n7	evad3rs	-
7.1-9.3.3 (4S-6S)	2014-	pangu, pangu8, pangu9 [111, 54]	Pangu Team	Extensively presented at hacker conferences such as BlackHat
8.1-8.4 (6)	2014	TaiG	TaiG	-
10.1-10.2 (6S)	2016-2017	yalu + mach_portal	Luca Todesco, Marco Grassi, Ian Beer[d]	Not intended for end-user use, usable by experts
11-11.1.2	2017	LiberiOS	Jonathan Levin	Designed for security experts, lacking an interface for general usability
11-13.5	2018	unc0ver	pwn20wned, Sam Bingner	Extensively maintained with many bug fixes, features, performance improvements, and counter-patches against Apple fixes
12-14	2019	checkm8 and checkra1n [101]	Luca Todesco *et al*	Also extensively maintained, working for iOS 14 beta even before its full release

Table 3.15: History of Jailbreaks on iPhone

[a](Model), alternate models omitted.
[b]Not an exhaustive list.
[c]Not affiliated with Apple.
[d]Developed exploits used, not directly affiliated.

Lock screen bypass. Lock screen bypasses tend to be induced exceptional cases discovered

seemingly often by end users who explore the lock screen interface beyond developers' intent. These bypasses are characterized by unexpected user inputs to some lock screen-available application (often a camera, clock, or weather application), and tend to require some care to be performed consistently. Table 3.16 catalogs a history of some notable lock screen bypasses.

Based on our evaluation, there is no evidence these bypasses are used by law enforcement to access personal data. However, as the phone is in a locked state, these bypasses serve as concrete indications of data which could be recovered without breaking encryption.

iOS	**Lock Screen Bypass**
4.1	Access to contacts and photos[a]
6.1	Access to contacts and photos[b]
6.1.2	Access to contacts and photos, initiate iTunes sync[c]
7 (beta)	Trivially access photos[d]
7.0.1	Access and share photos[e]
9.0.1	Access to contacts and photos[f]
9.2.1	Multiple bypasses to access the home screen, extent of compromise unclear[g]
12.0.1	Access to photos[h]
12.1	Access to contacts[i]
13	Access to contacts[j]

Table 3.16: History of iOS Lock Screen Bypasses

This list is not exhaustive. Drawn from various sources, primarily [112]

[a] https://www.forbes.com/sites/andygreenberg/2010/10/25/careful-iphone-owners-simple-backdoor-lets-anyone-bypass-password-protection/

[b] https://www.forbes.com/sites/adriankingsleyhughes/2013/02/14/ios-6-1-bug-allows-snoopers-access-to-your-photos-and-contacts/

[c] https://www.forbes.com/sites/adriankingsleyhughes/2013/02/18/new-ios-lock-screen-vulnerability-uncovered/

[d] https://www.forbes.com/sites/andygreenberg/2013/06/12/bug-in-ios-7-beta-lets-anyone-bypass-iphone-lockscreen-to-access-photos/

[e] https://www.forbes.com/sites/andygreenberg/2013/09/19/ios-7-bug-lets-anyone-bypass-iphones-lockscreen-to-hijack-photos-email-or-twitter/

[f] https://appleinsider.com/articles/15/09/23/ios-9-security-flaw-grants-unrestricted-access-to-photos-and-contacts

[g] https://seclists.org/fulldisclosure/2016/Mar/15

[h] https://appleinsider.com/articles/18/10/12/voiceover-bug-lets-hackers-view-iphone-photos-send-them-to-another-device

[i] https://www.theverge.com/2018/11/1/18051186/ios-12-1-exploit-lockscreen-bypass-security

[j] https://www.theverge.com/2019/9/13/20863993/ios-13-exploit-lockscreen-bypass-security

3.3.2 Local Device Data Extraction

Seizure of iOS devices has occurred in high-profile and common criminal cases alike. Motherboard (the technology branch of Vice news) published a database of over 500 cases involving iPhone unlocking, of which 295 included executed unlocking of iPhones either directly or through federal law enforcement or commercial partners [38, 40]. The San Bernardino shooting in 2015 famously sparked the "Apple v. FBI" case when the FBI sought access to a locked iPhone 5C [9, 11], and similar cases have occurred since [105].

In order to use forensic tools to the greatest extent, law enforcement officials seize iPhones along with any other devices which may aid in accessing those phones such as laptops and/or other Apple products [6, 113]. They isolate these devices, for example by placing them in Faraday bags, preventing remote wipe or control, and keep them charged to prevent shutdown (and thus eviction of encryption keys from memory), and in some cases to prevent iOS USB Restricted Mode [85] from coming into effect [6, 79, 113] using inexpensive adapters. USB Restricted Mode can also be bypassed in certain circumstances by booting the iOS device to DFU mode and delivering an exploit [80]. Devices are often examined by officials trained in the use of third-party forensic tools, or devices are sent to those third parties for examination [6, 7].

In the field, many devices are likely to be seized in the After First Unlock (AFU) state, having been unlocked once since their last boot but currently locked. This is immediately clear when considering the alternatives: if the device has been power cycled, it is very likely to have been used shortly after being powered on. In order to use most functionality, unlocking is required, and thus the device transitions to AFU. In this state, many decryption keys for data objects belonging to the AFU (or weaker) protection class will remain resident in memory, having been decrypted by the SEP [27, 28]. iOS will automatically use these keys to decrypt data requested by the iOS kernel on behalf of itself or of apps, including any forensic tools that can be installed without causing a device reboot.

Through companies including Elcomsoft [6], Cellebrite [7], and Oxygen [16], law enforcement agents can obtain forensic extractions of even the latest devices. These companies are naturally vague in the details they make public as they rely on exploits which must be unmitigated by Apple, however they make some information available in the form of advertisements and blogs [6, 7, 16]. Known exploits are also commonly used on older devices which no longer receive updates or which cannot be patched, such as with hardware/firmware vulnerabilities like checkm8 [16].[24] The common goal of extraction tools is to run an extraction agent on the device which reads any available files and exfiltrates them to an analysts device or computer. Sometimes, this takes the form of inducing a backup of the iOS device to the investigator's computer. This approach generally requires either a kernel compromise or running unsigned code, and thus a jailbreak, or exploiting/bypassing Apple's signing infrastructure (potentially among other exploits) to sign and execute such an agent [15].

In total, depending on the device (iPhone model), device state (e.g. AFU), types of seized devices, iCloud settings (e.g. 2FA enabled), and operational security (e.g. passwords on

[24]For more information on the checkm8 jailbreak, refer to §3.3.1

sticky notes next to laptop) of the target, law enforcement may obtain partial or complete access to extensive categories of data as listed in Figure 3.17. It is also clear that extraction of this depth and magnitude (all or most data categories listed) is historically the norm due to the extent of data accessed by forensic tools for over a decade as tested by DHS following NIST standards for forensic software testing [22, 42].

Figure 3.17: List of Data Categories Obtainable via Device Forensic Software

- Contacts
- Call metadata[a]
- SMS/MMS messages
- Stored files
- App data[b]
- Location data
- Wi-Fi networks[c]
- Keychain data[d] (authentication tokens, encryption keys, and passwords)
- Deleted data[e]
- iCloud authentication token(s)

Source: Elcomsoft [6] and Cellebrite [7] blogs and documentation, among others

[a]Participant phone numbers, call duration, etc.

[b]Particularly, data from applications which are not designed to opt into higher protection classes than AFU.

[c]Which can be used to determine location history [114].

[d]Some Keychain data which is configured into higher protection classes may be unavailable for extraction.

[e]Only in some cases when the full filesystem was extracted

Bypassing Data Protection. The encryption implemented in Data Protection is the last layer of defense against unauthorized device access, allowing devices to maintain data security even in the event that an attacker compromises the OS running on the device. It is up to app developers to opt-in to Complete Protection for sensitive data. However, this is not always done, even when critical for user privacy or confidentiality. For example, Elcomsoft discovered in 2020 that the iOS app for `mail.ru` (a Russian email provider and internet company) elected to put email authentication tokens into the iOS Keychain at the "Always Available" level of protection, meaning that these keys were entirely unprotected against physical or logical extraction [97]. Because there are no known practical attacks against modern encryption itself, current data extraction techniques (when the user is not available or willing to cooperate) tend to follow one of the following three approaches:

Accessing devices with keys available. The most straightforward approach to bypassing encryption is to simply obtain the device in a state where the necessary decryption keys are loaded into the device memory. Forensic tools such as the Cellebrite UFED (see images in Figures 3.18 and 3.19) or XRY Logical are able to connect to iOS devices and either initiate

a backup or otherwise request files and data over Bluetooth and/or a physical link via the Lightning port [19]. Data which is accessible by the iOS kernel (data for which keys have been decrypted and are available in memory) can be requested and exfiltrated directly. These forensic tools seem to work through a combination of proper use of iOS APIs and exploitation or circumvention of access controls in iOS. Forensic software companies sell devices which are as easy-to-use as possible [16] and then offer bespoke consultations for devices which are inaccessible using these more basic methods [7]. Refer to Figure 3.17 for a complete list of data categories seemingly regularly [38] extracted via forensic software.

Figure 3.18: Cellebrite UFED Touch 2

Source: Privacy International [19]

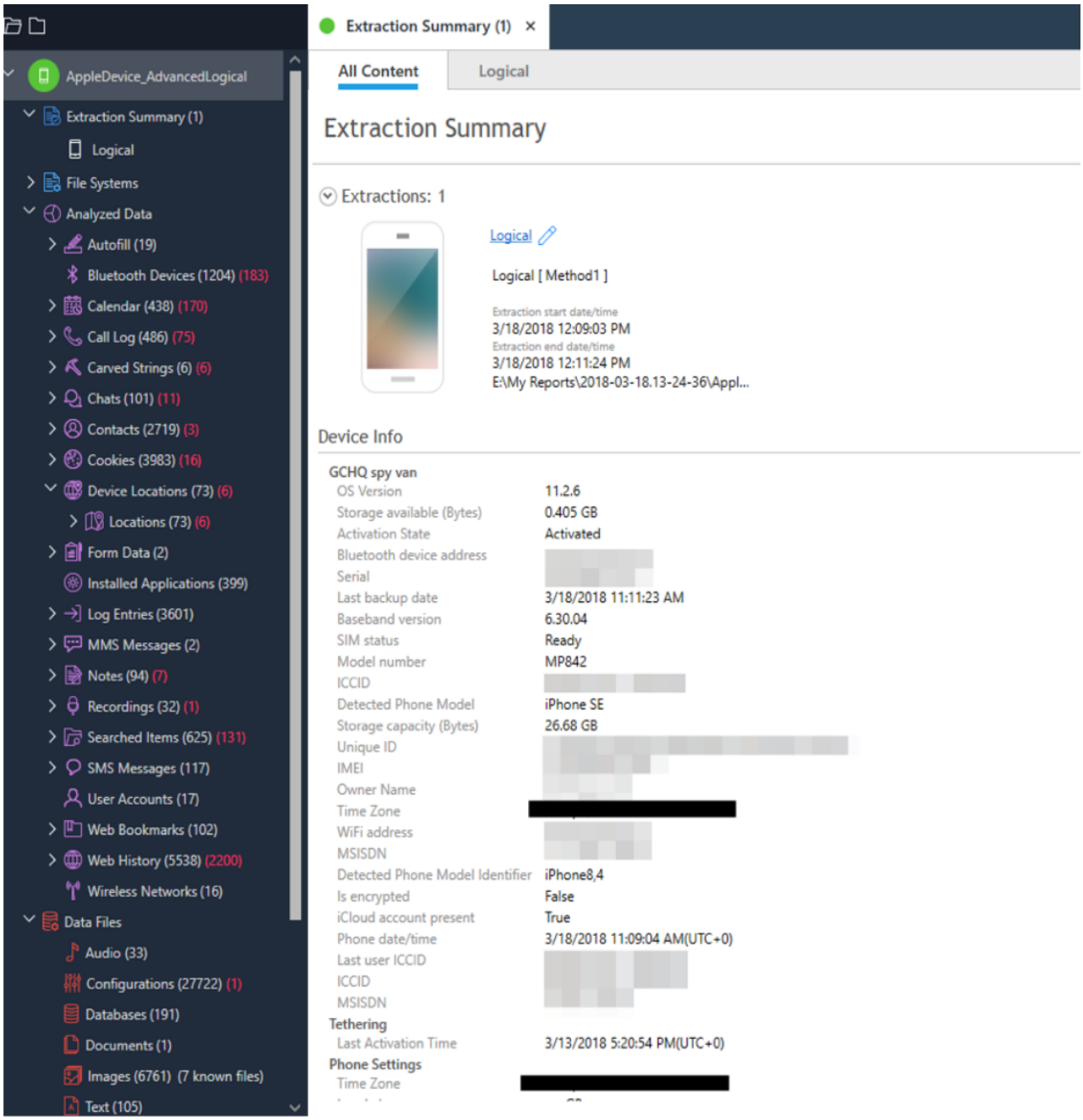

Figure 3.19: Cellebrite UFED Interface During Extraction of an iPhone

Source: Privacy International [19]

Bypassing protections. Data which is protected under Data Protection and not available for immediate extraction may still be accessed in some cases. GrayKey, for example, seems to have been able to extract user passcodes by exploiting the SEP to enable passcode

brute-force guessing [23, 104]. Documents obtained by Upturn [30][25] display records of law enforcement agencies gaining AFU and even BFU access to iPhones, presumably using Graykey. Figure 3.20 displays a small selection of these records.

iPhone X	6-Digit Pin	Before First Unlock	Password Recovery Completed
iPhone 7 (Plus)	4-Digit Pin	Before First Unlock	Password Recovery Completed
iPhone 7 (Plus)	6-Digit Pin	Before First Unlock	Password Recovery Pending
iPad (3rd generation)	4-Digit Pin	Before First Unlock	Password Recovery Completed
iPhone 6S	Alphanumeric	Before First Unlock	Password Recovery Completed
iPhone 8	6-Digit Pin	After First Unlock	AFU Extraction Completed

Figure 3.20: Records from Arizona Law Enforcement Agencies Documenting Passcode Recovery on iOS

Note passcode recovery on AFU and BFU iPhones up to X. Many rows omitted.

Source: Upturn [30]

An Elcomsoft blog article alleges that a brute-force passcode guessing strategy can only be conducted rapidly (many passcode attempts per second) if the device is in an AFU state, and that otherwise such guessing attacks require upwards of 70 days to brute-force a 4-digit passcode [102]. Once the user passcode is known, law enforcement access to user data is relatively unbounded [102], as the entire iOS filesystem, Keychain, and iCloud contents can be extracted, among other capabilities. For example, "Significant Locations" [115] (GPS locations which the iOS device detects are commonly occupied) could be extracted directly from the device using GrayKey in addition to the Cellebrite Physical Analyzer product in 2018 [41].[26] GrayKey in 2019 could even reportedly bypass USB Restricted Mode [19]. See also Figure 3.21 for images of the GrayKey device from the FCC ID filing [106].

[25]For policy analysis of these documents and far more, see Upturn's recent document entitled "Mass Extraction." [17].

[26]Significant Locations are end-to-end encrypted when syncing to iCloud [31] but this protection is irrelevant for data at rest.

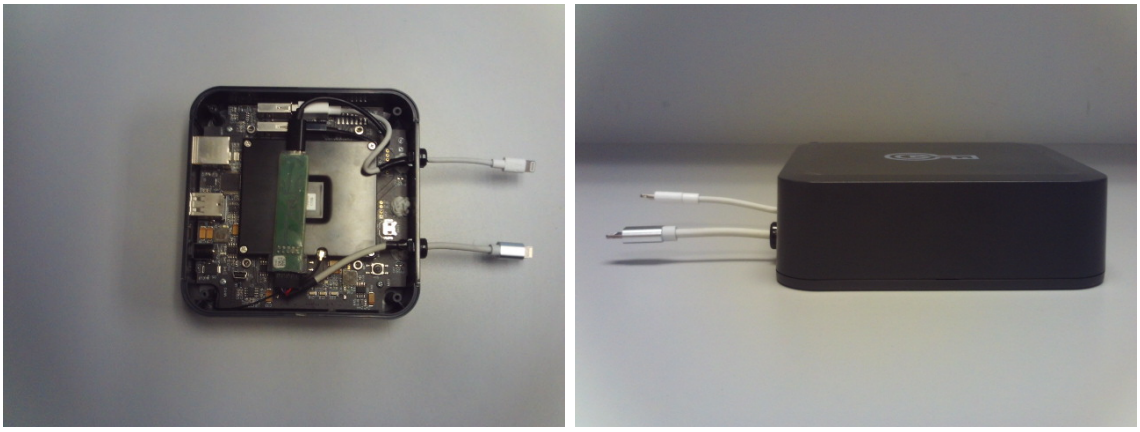

Figure 3.21: GrayKey by Grayshift

Left: Interior view. Right: Exterior view. Source: FCC ID [106]

Alternate data sources. In the case that forensic tools and even privileged (law enforcement only) consulting services such as those offered by Cellebrite and GrayKey fail to provide desired results, it is possible for investigators (or equivalently, hackers) to access iOS device data by other means. If a target's laptop can be seized in a search and accessed (perhaps with the password written nearby), access to iOS backups can render device extraction redundant. Even when encrypted on a local computer, iOS backups have been accessible to law enforcement due to a weak password storage mechanism enabling brute-force decryption via password cracking [84]. Further, forensic software companies offer cloud extraction services detailed in §3.3.3.

3.3.3 Cloud Data Extraction

iOS devices alone contain extensive personal data which law enforcement agents may seek to extract. While devices themselves are increasingly protected from such analysis (refer to §3.1), data is also increasingly stored or synchronized to the cloud [13, 97, 116, 117]. Thus, cloud service data extraction can replace or even surpass the value of mobile devices from the perspective of law enforcement. Apple complies with warrants to the extent they claim to be able to, and offer law enforcement access to various categories of data, particularly those stored in Apple's iCloud. For details, see Figure 3.12. However, law enforcement agents also use warrants to replace consent to access devices and services [38] and cut Apple out of the loop. Speculatively, the reasons for this could be reduced legal exposure, saving cost/time, or preventing potential notifications of targets.

Forensic software companies have followed this market and branched into developing cloud extraction solutions [13, 117]. While a full SEP exploit could extract the entire Keychain and thus authentication credentials for many cloud services, these tools commonly exploit iOS devices to extract cached authentication tokens only protected at the AFU class or worse, for services such as iCloud [27, 28], Dropbox, Slack, Instagram, Twitter, Facebook, Google

services, and Uber [117]. These tokens seem to be stored in AFU to prevent interruptions of service upon device lock, but this convenience poses a huge risk as it removes SEP protection, relying only on the security of the iOS kernel.

If other Apple devices are seized (laptops, Apple TVs, etc), they may be used to aid in authentication to iCloud, but commonly usernames and passwords can be acquired directly during search and seizure either through technical measures such as software tools [6, 97] or simply by searching for written passwords. Essentially, access can be incrementally increased in combination with other access in a compounding fashion. Figure 3.22 lists data categories potentially obtainable via cloud forensic extraction.

Figure 3.22: List of Data Categories Obtainable via Cloud Forensic Software

- iOS backups
- Internet search history on Safari or Chrome
- iCloud documents and app data
- Contacts
- Call metadata[a]
- Calendars
- Photos and videos
- Notes
- Reminders
- Find Friends and Find My data[b]
- Device information
- Payment card information
- Dropbox content
- Social media accounts and content (Twitter, Facebook, Instagram)
- Google services content
- Uber account and activity

Source: Cellebrite [13], Elcomsoft [116], and Privacy International [117]

[a] Participant phone numbers, call duration, etc.
[b] These services enable location tracking of Apple devices, whether the user's own or their friends [67].

3.3.4 Conclusions from Bypasses

The evidence above indicates the availability of current and historic bypass mechanisms for iOS protection measures. This strongly indicates is that, with sufficient time, money, and fortunate circumstance (e.g. capturing a phone in an AFU state), law enforcement agents can typically extract significant (if not all) personal data from modern iOS devices, despite Apple's claims around user privacy [10, 27, 28]. This is exacerbated by Apple's failure to widely deploy Complete Protection over user data, and its failure to more broadly secure cloud services (particularly, the decision to store cloud authentication tokens in AFU). These facts combine to offer extensive access to law enforcement agents, rogue governments, and criminals.

3.4 Forensic Software for iOS

For nearly as long as there have been iPhones, there have been forensic tools designed to circumvent the protection measures on those phones to enable law enforcement agents to access sensitive personal information in pursuing a case. Phone forensics is not new [118], but with the introduction of the iPhone in 2007 the amount of personal data aggregated onto one device that so many people began carrying every day increased massively. Additionally, it is critical to realize the accessibility of professional forensic tools such as Cellebrite's UFED [14], and even of individualized consulting services such as Cellebrite's Advanced Services [7] for unlocking phones. Law enforcement agencies, including local departments, can unlock devices with Advanced Services for as cheap as $2,000 USD per phone, and even less in bulk [17], and commonly do so [17, 40, 19].

For a complete list of the forensic tools tested by DHS, as determined by publicly-available reports as of this writing, and the data forensically extracted by those tools, see Appendix C and archived DHS reports [22]. Unfortunately, the NIST standard for testing devices is unclear as to whether the device should be in a locked state during testing [42]; we note, however, that certain categories of data seem inaccessible to forensic software in various cases, and as such we assume that this was caused by Data Protection. If false, then these tests simply document the extent to which the tested forensic tools support iOS data transmission and formatting. If true, then in summary these reports imply the following:

Successful extraction. In most tests, most or all of the targeted data (see Figure 2.1) is successfully extracted against the latest iOS devices. However, there are exceptions, and reports after January 2016 are less clear as discussed below. Simultaneously, documents acquired by Upturn demonstrate records of law enforcement access – as extensive as Before-First-Unlock passcode recovery – to all generations of iPhones [17, 30].

Relevance of Data Protection. The major category of such exceptions are tests against iOS versions which include updates to Data Protection (see Table 3.10). In some cases, forensic tools were unable to access certain data. Particularly, app data and files seem to have been successfully protected against forensics for a time in 2015-16, coinciding with an expansion of DP.

Limited software diversity. A small number of forensic software companies frequently iterate their products. This is demonstrated in the DHS tests as many Cellebrite, XRY, Lantern, Oxygen, MOBILedit, Secure View, and Lantern devices being tested between 2009 and 2019 (see Appendix C), and each generally successfully extracting data from contemporaneous generations of iOS devices over time [22].

Reporting inconsistencies. Starting in February 2016, the quality of the reports degrades notably, and it is unclear in many cases whether data was extracted from iOS and not displayed properly by the forensic software, or simply not extracted at all. Some reports[27] showed inconsistencies between the analysts notes on forensic performance and the summary

[27]For example see April 2017 "Electronic Evidence Examiner - Device Seizure v1.0.9466.18457."

tables in the final report (e.g. claimed extracted data in the notes, but this success not indicated in the summary table).

GrayKey. One notable exception to the vagueness of post-January 2016 reports was the GrayKey test in June 2019. GrayKey definitively extracted data covered by Data Protection on iPhones 8 and X according to the report.

Based on this information and a report of over 500 forensic access warrant filings and executions against iOS devices [38], it seems that law enforcement agencies are generally able to pay for or collaborate to gain access to iOS device, even of the latest generations. Particularly, federal agencies (FBI, DHS) are able to consistently extract data, and are consulted by local law enforcement agencies for such services if consent to access a device is not attained.

3.5 Proposed Improvements to iOS

iPhone users entrust the privacy of their communications and activities and the security of their accounts and data to Apple. They rely on Apple's responsiveness to vulnerabilities and mitigation of potential attacks to keep them safe from malicious actors, including hackers, potentially overreaching agencies, or rogue governments and corporations. However, as the history and current landscape of iOS protection bypasses show, Apple devices are not immune to compromise and their users are not completely protected. Impenetrable security is of course an impossible goal, but based on our analysis we have formulated the following recommendations for Apple software and hardware to mitigate channels of data extraction and improve iOS security.

Leveraging Data Protection. Data Protection on iOS is a powerful tool to ensure the encrypted storage of sensitive data. However, the AFU class is used as the default for third-party apps, and for many built-in data categories, notably including cloud service authentication tokens. iPhones are naturally mostly carried in an AFU state, and so this class of protection is in fact a liability given the availability of iOS forensic software devices. There are benefits to the AFU state: for example, keeping VPN authentication secrets available during lock [27, 28] prevents an interruption of connectivity, and keeping contacts accessible means that iMessages and SMS/MMS received while the phone is locked can display the sender identity (name) rather than just a phone number or email address.[28] Apple provides these features in iOS and therefore justifies their AFU classification. However, other potentially sensitive data, most notably cloud service authentication tokens, are also classed as AFU, putting them at risk even when not in use. We suggest two complementary approaches: first, that Apple thoroughly review the need for AFU in each case it is applied, and default more data into the CP class; and second, that Apple develop and deploy a system for runtime classification of data accesses in order to determine if and how often data is needed while locked. In such a system, data could be automatically promoted to CP if unused

[28]When the device is before first unlock and a text is received, the sending phone number is displayed, as tested by the authors with iOS 13.

during device lock. Dynamically increasing security based on app and user behavior would result in a system more widely protected without interrupting the user experience, which based on the amount of data currently classed as AFU is an important point for Apple in the design of iOS security. Apple has developed a powerful framework for protecting user data; the only limitation of Data Protection is that it is not applied in the strictest fashion possible.

Dynamic Data Protection. In addition to strengthening usage of Data Protection and particularly the Complete Protection class, we additionally recommend an intelligent system which learns from user and app behavior to predict when certain file keys will be used, and evict the encryption keys from memory of files which are unlikely to be imminently used. The advantages of this system would be minimal user experience interruption (with a good predictor, but with Apple's significant developments in machine learning hardware on iOS devices we believe this is feasible) in combination with increased protection against jailbreak access to values in memory. Such a system could fetch keys as needed, and potentially re-authenticate the user though FaceID if an access was considered suspect (anomalous), or a USB device had been attached. As the user will likely be looking at their screen during use, such FaceID re-authentication could be seamlessly woven into the user experience, or even be opt-in to prevent user frustration or confusion.

End-to-end encrypted backups. Apple continues to hold keys which are able to decrypt iCloud Backup data. However, there seems to be little value in maintaining Apple's ability to decrypt this data aside from recovery of certain data in case of complete device and account loss [65, 31]. Google offers and end-to-end encrypted backup service [119] for Android which, while imperfect (refer to Chapter 4), vastly improves the security of Android backups. Why Apple, a company which markets itself around user privacy [29], has not implemented a competing solution is an issue of curiosity and concern. Apple already has the infrastructure and design in place to implement such a system [31]. Keys could be held in iCloud Keychain, and backup data would thus be end-to-end encrypted, inaccessible to Apple but available to any trusted device the user has authenticated.

End-to-end encrypted iCloud content. Apple maintains access to photos, documents, contacts, calendars, and other highly personal data in iCloud. Similarly to backups, the infrastructure to place these data into CloudKit containers and allow them to be end-to-end encrypted amongst trusted devices (which can access iCloud Keychain to share encryption keys) would massively reduce the efficacy of cloud extraction techniques. If user data loss is a concern, recommend that users create both local and iCloud backups regularly, or automate this process. SMS and MMS messages are immediate prime candidates for this as the iMessage app (which manages SMS, MMS, and iMessage) already integrates with CloudKit.

Avoid special cases which bypass encryption. iMessage in iCloud uses an end-to-end encrypted container to prevent Apple from accessing iMessage content. However, this security is rendered moot when the encryption key is also included in an iCloud Backup to which Apple has access. Based on Apple documentation, it seems that this feature was included to provide an additional avenue for recovery of iMessages, but the security implications

are significant and as such this loophole should be closed. We thus recommend that Apple continue transitioning data into end-to-end encrypted iCloud CloudKit containers rather than holding encrypted data and the relevant keys.

Local backup passwords. Although Apple increased the iteration count for PBKDF2, the one-way cryptographic function which protects the password in a backup, to the point that most brute-force attacks are infeasible, local backups have two limitations. First, because of the nature of a local backup, guessing limits cannot be enforced. Thus, sufficiently high-entropy passwords must be chosen by users, which may not always be the case. User education and interface design to encourage strong backup passwords, or a system which involves an on-device secret could strengthen this mechanism. Additionally, in iOS 11, Apple removed the requirement to input the old backup password if a new one is to be set [120], and as such the passcode is sufficient to initiate a new backup. Although Apple goes to great length to protect the passcode, this represents a single point of failure where multiple layers of security could be used instead.

Strengthening iCloud Keychain. iCloud Keychain represented a significant step forward for the security of Apple's cloud services, enabling strong encryption with keys isolated even from Apple, assuming trust of the HSMs. That trust, however, creates a point of failure for the system: users have no way of ensuring that the HSM cluster they are backing encrypted data up to behaves in the way they expect. That is, as the device interaction with the HSM is limited to authentication via the SRP protocol and sending encrypted data, there is no authentication of the security or correctness of the HSM. Even if the user is enrolled with trustworthy HSMs initially, if their iCloud Keychain record is invalidated the user can simply re-enroll [31]; this is relevant as a targeted user could be intentionally invalidated, and re-enroll with a compromised server rather than an HSM. As Apple seeks to provide options for recovery of this data, it can't be encrypted with keys on-device (as these would be lost with device loss/failure) or with keys derived from the user passcode or passphrase (in case of forgetting those). iCloud Keychain data could potentially be encrypted with keys derived directly from the user biometrics, but this isn't without risk either. Another plausible solution for authenticating Apple HSMs to users could entail a Certificate Transparency-like [121] solution wherein peers validate the addresses and correctness of the HSMs and publicly broadcast or peer-to-peer share this information.[29]

Abstract control of the USB interface. USB Restricted Mode [85] was a step towards securing iOS devices against invasive forensic devices which operate by attaching to the Lightning port and transmitting exploits, data, and/or commands over the USB to Lightning interface. However, we observe that forensic software tools are continuing to exploit this interface [79] and further that the checkm8 jailbreak is unpatchable on iPhone X and earlier [101]. In order to address these issues, the iOS kernel should be able to interpose and manage the USB interface on iOS devices such that security controls can be enabled, cryptographically-secured device authentication can occur, and perhaps even intelligent

[29]This solution was informally proposed by Saleem Rashid, and is reminiscent of TACK, a previous Internet Draft by Moxie Marlinspike and Trevor Perrin [122].

systems which recognize commonly-used Lightning devices and evict encryption keys or authenticate the user when anomalies are detected. As part of increasing USB protections, debugging and recovery interfaces must be hardened as well, particularly to mitigate exploits such as checkm8.

Further restrict debugging and recovery interfaces such as DFU and JTAG. DFU mode, short for Device Firmware Upgrade, is a low-level bare-bones operating system which enables directly installing firmware patches. The cases in which this is required are limited, and ideally uncommon as this mode is intended entirely for remediating software problems. Users can place their own device in DFU mode, and exploits such as checkm8 can potentially exploit the capabilities of this mode [101]. In order to prevent such unpatchable exploits, DFU mode could require cryptographic authentication to access, for example using secure hardware on a trusted Apple computer owned by the user or even by Apple. The inconvenience of such a requirement would be offset by the rarity of its use.

Additionally, per the 2020 FCC filing by Grayshift of the GrayKey device [106], we observe what may be JTAG hardware in the device (as shown by the interior view images, refer to Figure 3.21). If this hardware debugging interface is vulnerable to unauthorized access, then further (ideally cryptographic, if possible) measures should be taken to secure this interface, particularly as end users will almost certainly have no need of it.

Increase transparency. Apple invests significant marketing effort into demonstrating their commitment to user privacy [9, 10, 26]. However, inconsistencies in Apple's approach to practically implementing privacy controls disempower users of iOS device. For example, user control over which apps can access contacts, calendars, and other built-in app data is a powerful yet understandable tool for managing privacy, but enabling or disabling the "iCloud" toggle in iOS settings can have dramatic privacy and even functionality implications which are relatively opaque. Increasing transparency and empowering users through safe default settings and informative, consistent interfaces would improve the usability and practical privacy experienced by iOS users.

Apple could integrate the extensive work of user interface researchers, particularly those who design for user empowerment and follow egalitarian principles. Design Justice [123, 124] is an ongoing area of research, industrial collaboration, and outreach which promote principles to foster equality and access in design. Opaque interfaces, such as the discussed controls surrounding iCloud and their pertinence to privacy, could be redeveloped with the Design Justice Principles in mind.

Leverage research constructions. In certain cases, functionality and security can create contradictory requirements for a system. For example, it could be a requirement that location data remain private and on-device, never synchronized to iCloud, but Apple could still seek to provide an iCloud service which allows location-based alerts. In such cases, where user privacy and features collide, constructions such as Secure Multiparty Computation (SMC/MPC) could be leveraged to, for example, allow iCloud and the device to privately compute a shared set intersection of user locations on-device to enable such alerts, or to enable some other such functionality. The research literature has provided extensive options in the form

of cryptographic constructions which can enable cloud functionalities without risking user privacy.

Leverage the community. Billions of Apple devices exist in the world, and so iOS is extremely widely used [46]. This worldwide community includes academics and professionals, students and experts, journalists and activists. Apple, then, has potential access to immeasurable knowledge of the experiences and needs of their users. With the bug bounty program [47], and through accepting externally-reported vulnerabilities in general, Apple began connecting with this community to improve the security and privacy of their products, and is continuing to do so with the Security Research Device program [125]. Leveraging the wealth of ongoing research, and embracing and implementing academic constructions can improve the security and privacy of Apple products as far as the state of the art allows. To facilitate this research, Apple could open source critical components (filing patents to protect their intellectual property if needed/applicable) such as the SEP and reap the benefits of allowing this network of academic and professional researchers unfettered access to help improve the devices they use and rely on every day.

Chapter 4

Android

Android is the most popular smartphone platform in the world, with 74.6% of global smartphone market share as of May 2020 [126] spread across over a dozen major device manufacturers [127]. The technical and logistical challenges in securing Android phones and protecting their users' privacy are numerous, and do not fully overlap with other segments of personal computing. The Android operating system is based on the open source Linux codebase, and so derives the benefits and risks of this underlying OS as well as new risks created by the mobile-specific features included in Android. Android's dominant market share also means that the impact of a new security vulnerability is acutely felt around the world.

The base platform of Android, the Android Open Source Project (AOSP), is a collection of open-source software developed by the Open Handset Alliance [128], an entity that is commercially sponsored by Google. AOSP defines the baseline functionality of the Android operating system. However, the Android ecosystem is in practice more complex, due to the fact that most commercial Android phones also incorporate proprietary Google software known as Google Mobile Services (GMS) [129]. GMS includes proprietary APIs and software services, along with core Google apps such as Chrome, Drive, and Maps, and the Google Play Store, which is used for app distribution [130]. There exist non-Google forks of Android such as Amazon's FireOS [131], but most of the Android devices in the United States [33] are manufactured by Google's partners [34] and use GMS. For the purposes of this chapter, we will consider "Android" to be the combination of the Android Open Source Project and Google Mobile Services.

4.1 Protection of User Data in Android

Android devices employ an array of precautions to protect user data: user authentication, runtime verification, data encryption, and application sandboxing. This overview is based on documentation released by the Android Open Source Project [132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150].

User authentication. Android provides numeric passcode and alphanumeric passphrase

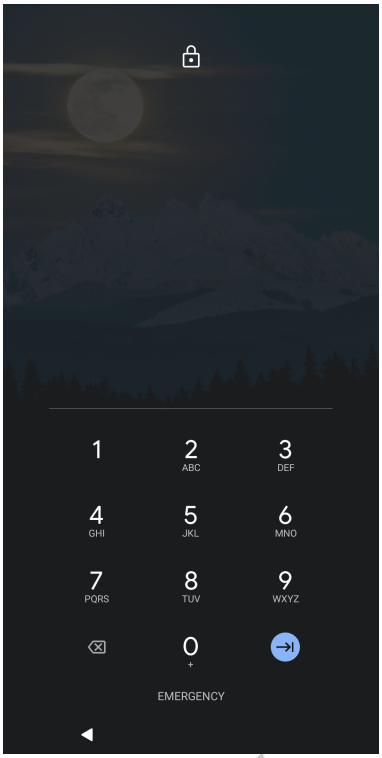

Figure 4.1: PIN Unlock on Android 11

authentication options for users, using an interface illustrated in Figure 4.1. Android additionally provides an authentication mode known as *pattern unlock*, wherein a user authenticates by drawing a pattern on a grid of points [135]. The security of the latter scheme has been called into question [151], with one academic study noting that effective security may be equivalent to a 2 or 3-digit numeric passcode [152]. Many Android devices also deploy biometric authentication sensors; fingerprint authentication in particular is natively supported by the AOSP operating system [139].[1] A new biometric authentication feature, Face Authentication, is currently under development in AOSP [153]. Android biometric authentication is deployed by manufacturers, and is not required by the Android OS.

Android devices support an additional set of authentication features known collectively as *Smart Lock*. These use information about a phone's environment as criteria to keep it unlocked under certain conditions. Critically, this is a method of *keeping devices unlocked*, which is distinct from a method of *unlocking a locked device*, which can only be done with a PIN/password/pattern or a biometric. For example, Smart Lock will keep a device unlocked while it is on-body or in a specific location (such as at home) [154]. One of the Smart Lock features, Trusted Face (formerly known as Face Unlock), allowed users to use their face (via the device camera) to unlock a device. This feature was removed in Android 10 [155], as it was easy to fool using static images [156], and is being replaced with the more general Face

[1]Most fingerprint readers are implemented either as external capacitive sensors or within the screen of the device.

Authentication biometric framework [153].

Application sandboxing. The Android systems utilizes application sandboxing as a mechanism to prevent apps from compromising each other or the OS as a whole. The basic sandboxing mechanism is provided through discretionary access control (DAC): files have permissions based on Linux user IDs, which are enforced in the Linux kernel. Each Android app has its own Linux user ID, and by default only has access to its own files, so any attempted direct access to the files of other apps or the core OS will result in an error. DAC is enforced by the Linux kernel [133]. Theoretically, bypassing this mechanism would require a kernel exploit in order to escalate privileges. However, the Android OS provides many other avenues for inter-application communication, which creates further opportunities for privilege escalation [157].

To enhance the limited security provided by DAC, Android also employs Security Enhanced Linux (SELinux), which allows granular control of the privileges granted to an application [146, 157]. For example, in the traditional DAC model, a compromise of a root-owned process[2] would be tantamount to compromise of the whole system. With SELinux, polices are configured so that a process is allocated only the specific privileges it needs to function; this is known as mandatory access control (MAC). In this setting, compromise of a root-owned process would not necessarily lead to full system compromise, since the kernel would prevent the process from performing actions outside of its capability set [158]. In Android, SELinux is enforced between the system and individual apps: apps are confined to the permissions available in their domains (process groups), and any violations result in an error at the kernel level [146].

Encryption. There are two forms of user data encryption on Android: full-disk, a direct parallel to Linux disk encryption, and file-based, which enables more granular categorizations of encrypted data.

Full-disk encryption is the original encryption mechanism on Android, introduced in Android 4.4 [141]. It is based on the Linux kernel's `dm-crypt` module [159], and uses a master key to encrypt block devices with AES-128 in CBC mode with ESSIV [141]. The master key in turn is encrypted using a key derived from the user authentication credential: a numeric passcode, alphanumeric passphrase, or pattern. However, because the same key is used for the entire disk partition, Android requires the master encryption key to be available on boot in order to activate core system functionality: specifically, subsystems such as telephony, alarms, and accessibility services remain unavailable until the user unlocks their phone. Once entered, the master key is never evicted from device memory, as it is needed for *all* block device operations on the data partition [141].

File-based encryption is a more recent development, and provides more granular control than full-disk encryption. The underlying mechanism for this encryption is the Linux kernel's `fscrypt` module [160], and uses AES-256 in XTS mode for file data encryption, and CBC mode to encrypt file metadata such as names [141]. File-based encryption segregates user

[2]"Root" in the context of traditional Linux/Unix refers to a special user account that enjoys super-user privileges over the entire system.

files into two categories: *Credential Encrypted* (CE) and *Device Encrypted* (DE). CE data is encrypted using a key derived from the user's passcode, and is thus available only after the device is unlocked. DE data is based only on device-specific secrets and is available both during the initial boot and after device unlock [161]. Because of this separation, file-based encryption allows the user to access important functionality prior to user authentication. CE is the default for all applications, and DE is reserved for specific system applications (Dialer, Clock, Keyboard, etc). Filesystem metadata (such as directory structure and DAC permissions) is not covered by this mechanism; a different encryption subsystem, metadata encryption [143] encrypts this information. The combination of file-based encryption and metadata encryption protects all content on the device as a result.

Outside of the CE and DE storage in file-based encryption, Android has no analogue to the iOS *Complete Protection* protection class: keys remain in memory at all time following the first device unlock [161].[3] Thus, an attacker who can logically gain access to the memory of a device following boot and first unlock can directly access encrypted data and keys.

Mandatory application of an iOS-style Complete Protection would be beneficial to privacy: this would reduce the possibility of live acquisition of encryption keys for a device. On the other hand, it would create large development and usability problems for Android phones and existing applications. Currently, the Android application lifecycle [162] distinguishes processes in a hierarchy of four categories based on importance: from high to low importance, those in the foreground (actively shown), those visible but in the background, those handling a background service, and those that are not needed anymore and can be killed. The current lifecycle does not incorporate the device's "locked" state into the paradigm, which means that an encryption scheme that evicts keys on lock may require a retooling of the Android process lifecycle. Enabling CE/DE storage itself required such a modification to application logic [161], and it is unclear how complex the additional changes would be in order to achieve complete protection as an option.

Removable storage. Android has first-class support for expandable storage, such as SD cards; Android allows applications to install to and hold data on an SD card. Media, such as photos and videos, can also be stored on the SD card. Originally, the "external storage" directory was considered to be removable storage – Android would have limited control over the access to files on the SD card (since the card could be removed anyway). Android 6 introduced adoptable storage [132], which allows the system to consider the SD card as a part of its own internal storage, which in turn allows the system to perform encryption and access control on the external medium. The downside of using this mode is that the SD card becomes tied to a specific device; since encryption keys are stored on the device, the SD card cannot be trivially moved to a different device [132].

Trusted hardware. Many security features are enforced by hardware mechanisms in modern Android phones. This is frequently implemented by through the use of a Trusted Execution Environment (TEE), which is a dedicated software component that runs in a special isolated mode on the application processor. This component enables, among other features, the secure

[3]This makes Android CE protection equivalent to Apple's AFU protection class, whereas DE protection is analogous to Apple's not protected (NP) class.

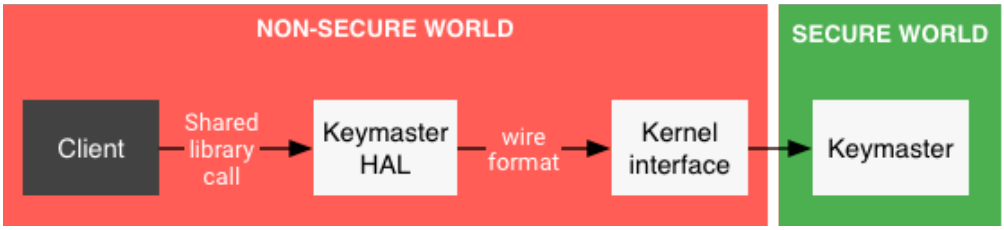

Figure 4.2: Flow Chart of an Android Keymaster Access Request

Source: Android Open Source Project Documentation [142]

storage and processing of cryptographic secrets in a form that is logically isolated from the operating system and application software.

Because the Android operating system is developed separately from the actual hardware on which it runs, Android cannot guarantee the presence of hardware support for the protection features it implements. In practice, a large majority of Android phones use the ARM architecture [163], and these system-on-chip (SoC) models (with ARMv7-A, ARMv8-A, and ARMv8-M [35]) include a mechanism called ARM TrustZone. Generically, ARM TrustZone partitions the system into two "worlds", each backed by a virtualized processor: a *secure world* for trusted applications, and a *normal world* for all remaining functionality. The secure world portion is used as a trusted execution environment for computing on sensitive data, such as key material [164]. Unlike Apple's SEP, TrustZone makes use of the same processor hardware on the SoC as is used by the Android OS and applications, and does not utilize a co-processor for trusted operations, but still provides a number of security properties for the Android system.

In particular, three standard trusted applications (TAs) are defined by AOSP to run in an Android TEE: Keymaster, Gatekeeper, and Fingerprint. The Keymaster TA is the backing application for Android's Keystore functionality, which allows apps to load, store, and use cryptographic keys. Clients outside of the TEE communicate with Keymaster through a hardware abstraction layer (HAL), which then passes the request to the TEE via the kernel; this flow is seen in Figure 4.2. The Gatekeeper and Fingerprint TAs are used to authenticate to the Keymaster, showing that a user has enter the correct passcode, password, or pattern (Gatekeeper) or has provided a correct fingerprint biometric (Fingerprint) before a Keymaster request is serviced [142].

Some Android manufacturers have begun to include additional dedicated security processors. For example, Samsung includes a Secure Processor in its Galaxy S line [165] and Google's includes a Titan M chip in its Pixel line [166]. This type of module, known as a *secure element*, is an analogue to the Secure Enclave Processor in iOS: a dedicated cryptographic co-processor, separate from the primary processor, for computation on secret data. In these systems, Keymaster keys are offloaded to the secure element, and all key-related actions occur off the main processor [144]. This offloading is in an effort to improve the security of key material, thus reducing the risk of e.g. side channel attacks on a TrustZone TEE [167, 168, 169].

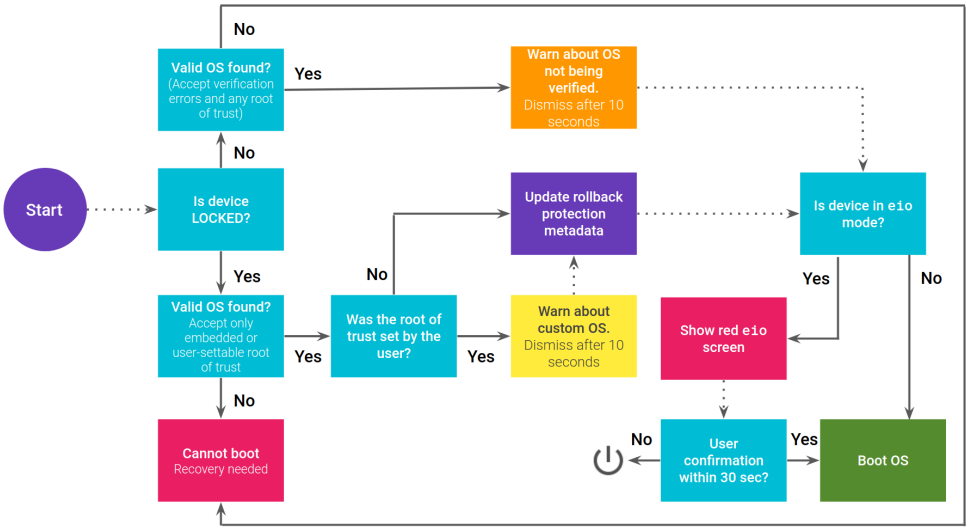

Figure 4.3: Android Boot Process Using Verified Boot

Source: Android Open Source Project Documentation [170]

Android Verified Boot. Android can optionally use the TEE to validate the integrity of the initial boot of the system, via its Android Verified Boot (AVB) process. The verification module, known as `dm-verity`, is designed to verify block devices using a cryptographic hash tree. Each hash node is computed as the device mapper loads in block data from a partition, with a final tree computed [137]. If this root hash value does not match an expected value stored in trusted hardware, verification fails and the device is placed into a non-functional error state. AVB additionally uses tamper-evident storage to prevent software rollbacks i.e. use of a prior version of Android that has vulnerabilities but is otherwise valid and trusted by the hardware root [149]. Figure 4.3 depicts the AVB boot process.

AVB can be disabled via a process known as "unlocking." An unlocked bootloader does not perform `dm-verity` checks on the `boot` partition, allowing untrusted code to execute on device boot [136]. This has legitimate use cases: for instance, unlocking can be useful for developers of Android modifications, and for users who wish to gain root access to their phones for additional customization and control. More information about rooting and bootloader unlocking can be found in §4.3.1.

Google Mobile Services. Thus far we have described options and security features that are directly available in the AOSP. However, many additional security features are tied to Google Mobile Services (GMS), which requires a software license from Google and is not part of the AOSP. GMS includes a number of Google-branded apps, including Drive for storage, Gmail for email, Duo for video calls, Photos for image storage, and Maps for navigation, as well as the Google Play Services background task, which implements additional APIs and security features. AOSP has some equivalents of these apps – such as generic "Email" and "Gallery" apps – but the GMS apps are generally higher quality [171] and are directly

supported by Google, with updates delivered through the Play Store [129]. A device that is approved to use GMS and passes compatibility tests is known as "Play Protect certified" [172]. Several manufacturers, like Samsung, LG, and HTC, partner with Google to have GMS included [34]; the few exceptions include those for business (Amazon FireOS [131]) and geopolitical (Huawei [173]) reasons.

Google services utilize the Google Cloud, which enforces network encryption (TLS) for communications between a client and the server. All data stored in the Google Cloud is encrypted at rest using keys known to Google [174, 175]. Core Google services such as Gmail (for email), Drive (for files), Photos (for pictures), and Calendar (for scheduling) also store data on the Google Cloud. This implies that data backed up using Android Auto-Backups (which use Drive) along with some forensically-relevant pieces of user data (as defined in Figure 2.1) are available to Google. Currently, the only GMS app that is known to use end-to-end encryption is Google Duo [176, 177], which is used for real-time video communication, though Google Messages may add end-to-end encryption soon [32].

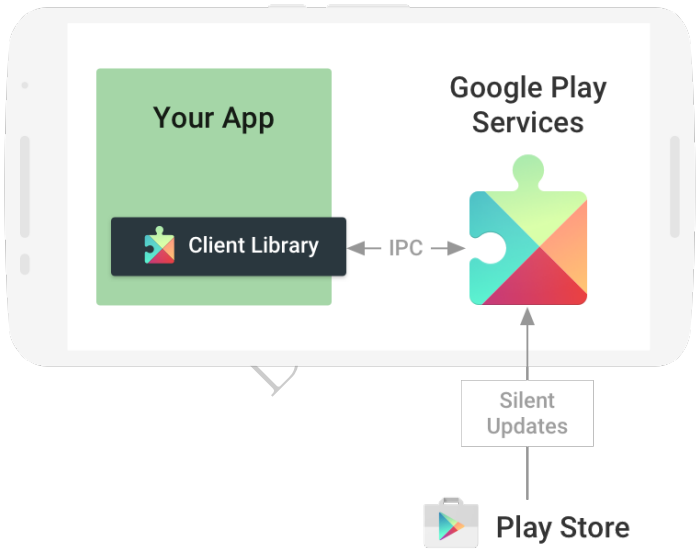

Figure 4.4: Relationship Between Google Play Services and an Android App

Source: Google APIs for Android [178]

Application developers integrate with GMS via the Play Services APIs. Any Android app that wishes to integrate with the GMS on Android must do so through Play Services, which persistently runs in the background [178]. App access to Google Account information, Mobile Ads, and Location information, among others, are all handled through Play Services [179], as seen in Figure 4.4. Of particular note is the SafetyNet API [180], which is a validation service for Android phones. Play Services monitors a device, and records whether an Android system is rooted or potentially compromised by malware; developers can then request this information for a device. Applications can chose to disable access to certain features based

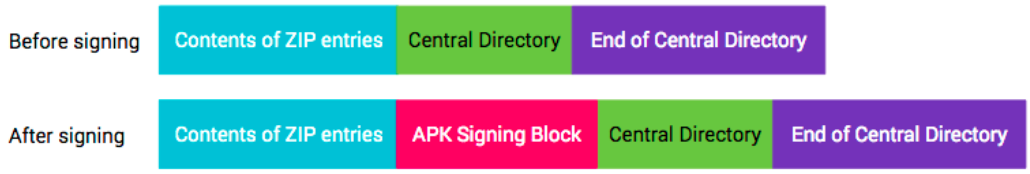

Figure 4.5: Signature location in an Android APK

Source: Android Open Source Project Documentation [185]

on the result of the SafetyNet check [181, 182]. For example, Google Pay, an electronic payment platform akin to Apple Pay, is disabled on devices that do not pass SafetyNet validation [183, 184]. An app for a non-GMS Android device (e.g. for Amazon Fire OS) cannot use these features, as Play Services is only available through a GMS license with Google [129].

Package signing, app review, and sideloading. AOSP Android requires that any application that runs is signed by its developer [186]. The actual mechanism involves verifying that the signature over the APK (Android PacKage) binary matches the public key distributed with the binary. The APK Signing Block contains all of the information required to validate the binary: a certificate chain and tuples of (algorithm, digest, signature). This block is embedded in the binary distributed to devices, as in Figure 4.5. Starting with Android 11, Android also provides support for `fs-verity`, Linux kernel feature that allows continuous verification of read-only files on a filesystem at the kernel level [187]. In Android, `fs-verity` is used to continuously validate APK files beyond the initial signing check [140].

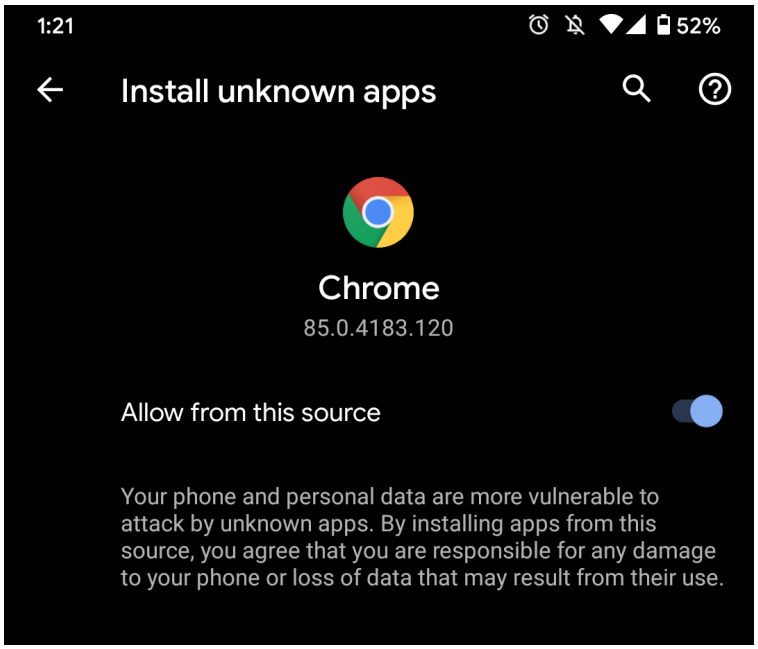

Figure 4.6: Installing Unknown Apps on Android

Allowing Chrome to install apps from unknown sources, also known as "sideloading."

The AOSP signature checks do not verify if the developer is legitimate or approved by Google, but simply acts as an authenticity check on the distributed binary itself. Google reviews apps submitted to the Play Store [188], and Android devices that use GMS incorporate an additional check to ensure that an application is downloaded from the Google Play Store [172, 189]. GMS-enabled Android phones have additional the option of installing apps from "unknown sources" by modifying a setting [190]. This allows users to "sideload", or install apps that are not distributed on the Play Store. Sideloading is required for the installation of applications from alternative (non-Google approved) developers or storefronts [191, 192]. Figure 4.6 illustrates an example of this setting, allowing Chrome to download and install unknown apps. This feature is off by default, and enabling it comes with a stern warning in the system, as applications distributed outside of Google's approved channels can be a vector for malware [193, 194, 195].

Backups. GMS provides transport for application data backups for Android devices. Application developers can choose to have their application's data backed up in either of two ways. The original method, available since the GMS in Android 2.2, is a Key/Value Backup (also known as the Android Backup Service) [196]. As the name implies, it allows apps to backup key and value pairs corresponding to a user's configuration. Use of Key/Value Backup is opt-in, and must be manually configured by the application developer [197]. The GMS in Android 6 introduced Android Auto-Backup [198], which automatically synchronizes app data to a user's Google Drive. This service requires no configuration from the application developer; developers must opt-out of the service on behalf of their users [197]. Both types of

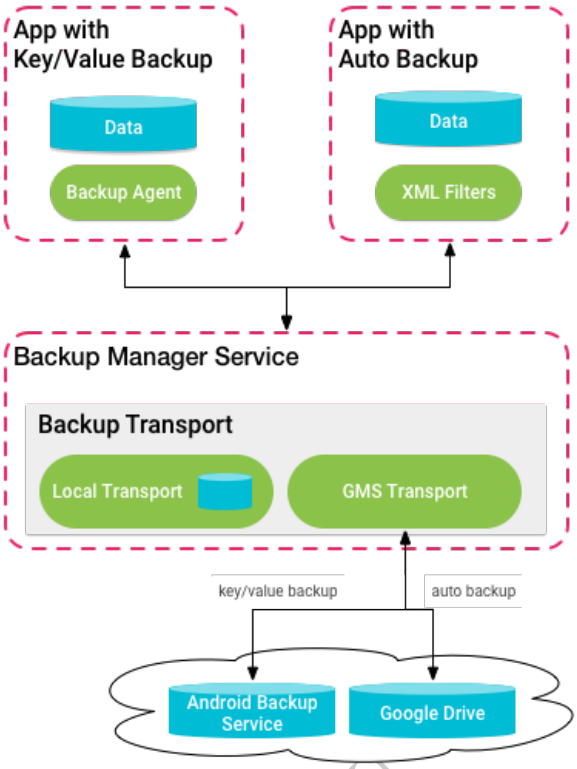

Figure 4.7: Android Backup Flow Diagram for App Data

Source: Android Developer Documentation [199]

backups – Key/Value Backup and Auto-Backup – are stored on the Google Cloud [199]. The transfer of a backup from a device occurs via GMS [199]. Data flows for these two methods are summarized in Figure 4.7.

Since 2018, Google has provided end-to-end encryption for application data backups stored on the Android Backup Service (Key/Value Backup data) [119]. Devices can generate an decryption key on-device for its backups, and use a user's authentication factor (PIN, password, or pattern) to encrypt this key. To protect this key, Google deploys a Google Titan HSM [200] as part of Google's Cloud Key Vault service [201], and device-generated backup encryption keys are held by this device. The Titan HSM is configured to reveal the backup key when presented with the correct authentication from the client (in the form of user's passcode). It also provides brute-force attack prevention and software rollback prevention. However, this end-to-end encryption is only provided for the opt-in Key-Value Backup service [119]. We find no indication that the automatic Android Auto-Backup service is covered by these guarantees, and it is additionally unclear how many developers manually opt-in to the Key-Value service rather than simply using Android Auto-Backup.

These backup services operate over application-local data only; other forms of personal data on an Android device are separately backed-up to Google services, as shown in Figure 4.9. This data is stored on the Google Account associated with a device, synchronized to Google

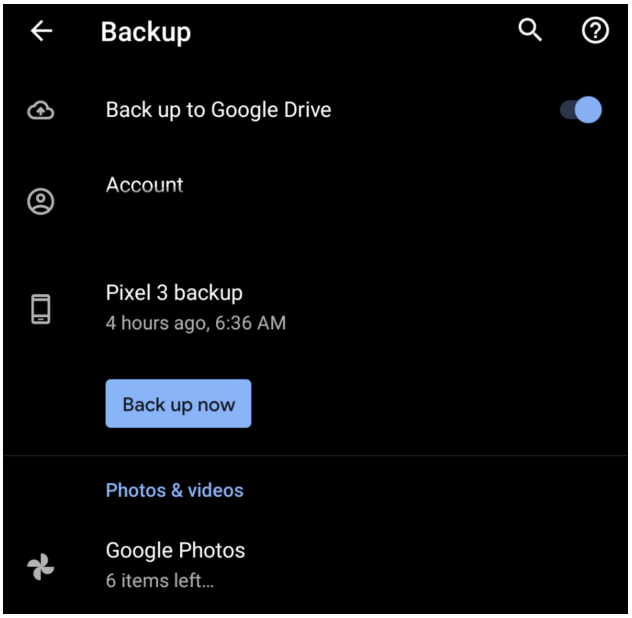

Figure 4.8: Android Backup Interface

Drive (similar to the data in Android Auto-Backup). Android support documentation [202] mentions that this data is encrypted, either with Google Account password information or with the user authentication factor (PIN/password/pattern). Google has access to encryption keys derived from the user password. However, the documentation is not specific as to what data is or is not covered by the different encryption methods. Refer to Figure 4.8 for an example backup configuration for a Pixel 3 running Android 11.

Some device manufacturers implement their own backup services [203, 204, 205]. AOSP provides password-protected backups [206] to a local computer via the Android Debug Bridge [207], Android's USB debugging tool. There are also a number of third-party backup tools [208, 209, 210].

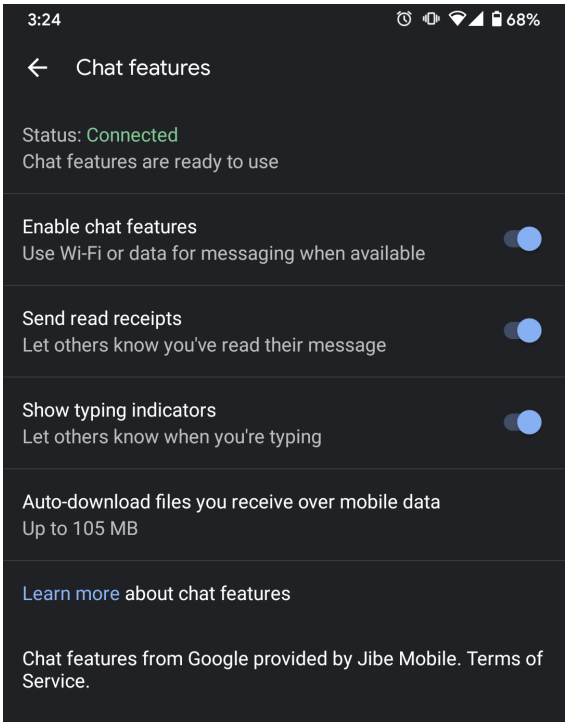

Figure 4.10: Google Messages RCS Chat Features

Figure 4.9: List of Data Categories Included in Google Account Backup

- Contacts
- Google Calendar events and settings
- SMS text messages (not MMS)
- Wi-Fi networks and passwords
- Wallpapers[a]
- Gmail settings
- Apps
- Display settings (brightness and sleep[b])
- Language and input settings
- Date and time[c]
- Settings and data for apps not made by Google (varies by app)

Source: Android Support Documentation [202]

[a] In general, photos and videos can be backed up to Google Photos, a separate Google offering.
[b] It is not clear from the documentation what other display settings are stored.
[c] The documentation does not specify what exactly this category entails.

Messages & Duo. Android provides support for standard carrier-based SMS and MMS messaging with a front-end provided from AOSP. While information is encrypted on-device [141, 138], these messages are not end-to-end encrypted in transit and are thus

vulnerable to both eavesdropping and spoofing attacks [211, 212]. Moreover, carriers maintain copies of both metadata and content on their back-ends [213]. The GSMA carrier associate has developed the more modern Rich Communication Services (RCS) [214] as an alternative to legacy SMS/MMS. RCS is a communication protocol that adds features such as typing indicators and read receipts to standard text messages. Individual providers (such as a carrier) deliver messages to each other on behalf of their customers [215]. Google Messages, a part of the GMS, supports RCS on Android under the term "Chat features", as seen in Figure 4.10. Google also provides Jibe Cloud, which allows users that do not have an carrier support for RCS to use it [216]. RCS support is currently unavailable in AOSP alone, although work is being done to provide system-level support for it [217]. In terms of security, RCS communication is secure between individual service providers [215] but is not end-to-end encrypted [218, 215], but developer builds of Google Messages provide evidence that Google is preparing for such an enhancement [32].

Google Messages also has a web client, wherein messages pass through from the browser through Google's servers to the user's device for sending [219]. Our testing shows that the web client for Messages does not work without a connection to a phone. We turned on Airplane Mode on a Pixel 3 device running Android 11, and noted that the web client was unable to send messages. The error message shown on the web client is found in Figure 4.11. There is no mention of end-to-end encryption between the browser and the device in the documentation, which implies that Google is able to see a user's messages as they pass through Google's infrastructure. Google claims to not keep any RCS messages on their backend [215] via data retention policies.

Duo is a GMS-provided video calling app that allows for end-to-end encryption by default. Devices participating in a Duo call perform key-exchange with each other to create a shared secret, and use that to encrypt the audio and video streams of the data [177]. In a one-on-one call, calls are peer-to-peer, i.e., audio and video data from one device is sent to the other without passing through a server in the middle; only call setup is routed through a Google server initially. This flow is shown in Figure 4.12. For group communications (and for one-on-one calls that block a peer-to-peer connection), calls are routed through a Google server. Audio and video is still end-to-end encrypted (via pairwise key-exchange between each of the participants), but the server learns metadata about the participants as it sets up the public keys and routes the encrypted communication between the participants [176].

4.2 History of Android Security Features

The previous section highlights key concepts in the current state of Android device security. Android has evolved significantly, however, from its original release in 2008, with major revisions of the operating system adding additional security features. Table 4.13 highlights the security features added in each release of the AOSP. Appendix B goes into detail about each item.

Update adoption. Operating system software updates have been historically slow to propagate to end devices [220]. Although Android is developed by Google, manufacturers

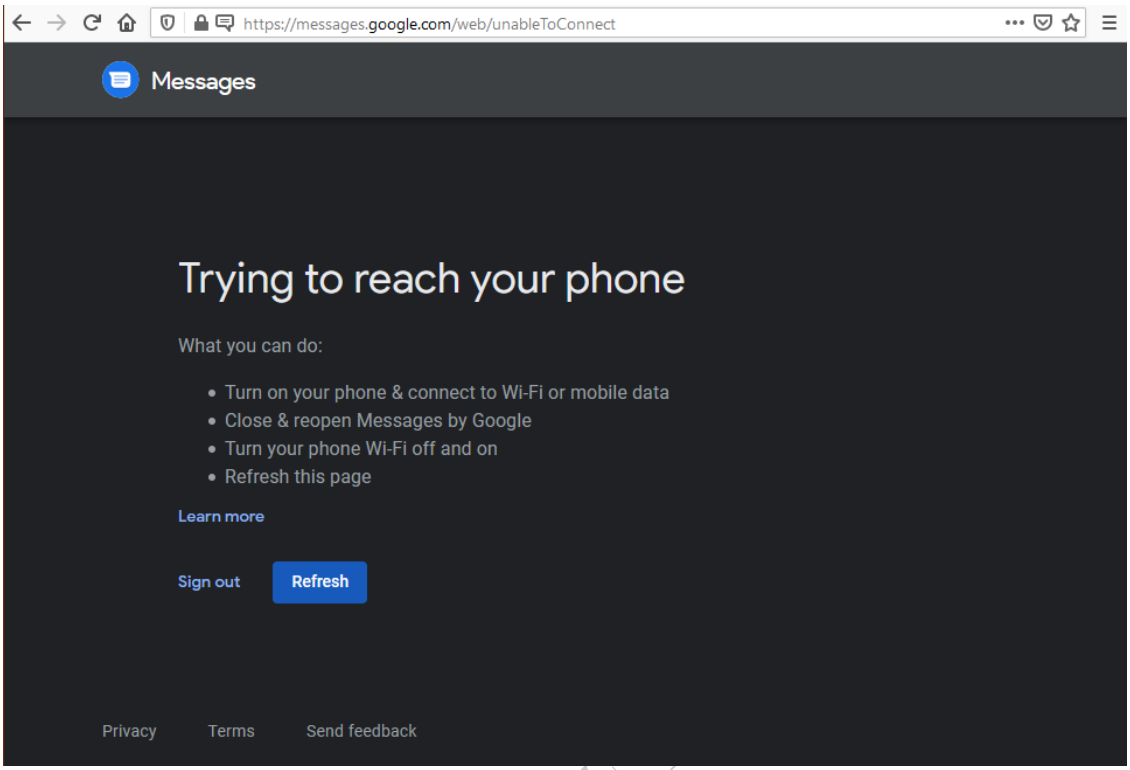

Figure 4.11: Google Messages Web Client Connection Error

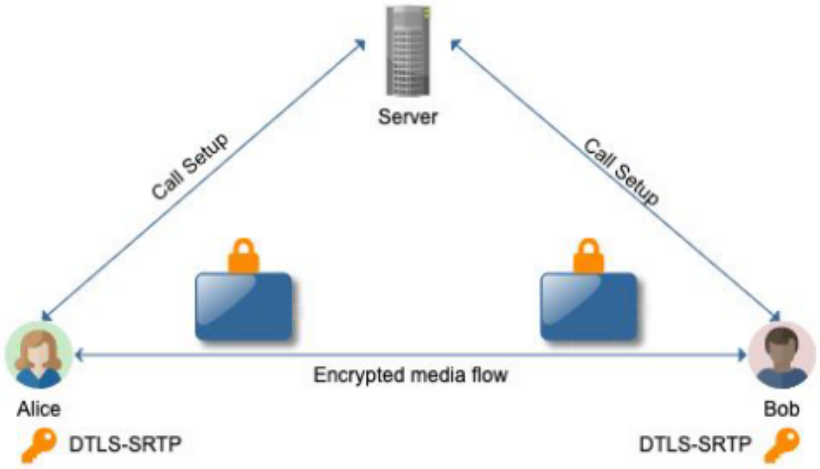

Figure 4.12: Google Duo Communication Flow Diagram

Source: Google Duo End-to-End Encryption Whitepaper [177]

often customize the AOSP source code [221], both to better match their target hardware and to add vendor-specific features. For example, Samsung modifies Android to provide its Bixby virtual assistant on its Samsung Galaxy series of Android devices [222]. These modifications introduce added complexity to the development of update images, which slows update adoption rates [223]. Compounding this is the shorter support lifecycle for Android. There is a broad array of Android phones, offered at different price levels. Those phones in the entry-level and mid-market segments rarely receive updates, if at all [127].

In an effort to reduce this additional hurdle to update deployment, Android 8.0 (2017) introduced Project Treble [220], which allows for updates to the Android OS without changing the underlying vendor implementation. Treble decouples the Android framework from the device-specific software that runs on a smartphone. This has improved update adoption rates [223], especially in the United States [224]. Update adoption still lags worldwide; as of May 2019, 42.1% of users worldwide were running version 5.0 (2014) or earlier, translating to around one billion Android devices in use that do not receive security updates [225].

4.3 Known Data Security Bypass Techniques

Android's large user base has created a market for exploits that bypass its security features. Even though Android's security has improved over the years (see Table 4.13), bypasses continue to be developed both in public and by private groups. The payoff for these bypasses can be substantial: for individuals and groups, large sums in the forms of bug bounties (both from Google [226] and others [48]), and for governments and law-enforcement agencies, significant access into the lives of individuals under investigation.

Rooting and software exploitation. Software exploits form the core of bypass techniques on Android, especially given the open-source nature of the AOSP. The usual end-goal of this category of exploits is to attain superuser access, a process known as "rooting". A rooted device can be used for non-malicious customization or development [227], but access to superuser capabilities allows for near-total control of a device, making it attractive for forensics [228]. Non-rooting software exploits include compromises of individual applications and capabilities that bypass the trusted hardware capabilities on a device, among others. These exploits are discussed in Sections 4.3.1 and 4.3.2.

Data extraction. Forensic examiners have several strategies to perform security bypasses. If an examiner has access to a user's password already, none of the Android protections matter and they can typically extract all of the data from both the device and the Google cloud. An examiner that actually has to perform a local device bypass can either do so manually, using available exploits on the internet, or they can do so by contracting the work to a forensics company. For cloud data, law enforcement can either procure information through a legal process with Google or by accessing Google services with tokens extracted from a local device. Local data extraction techniques are discussed in Section 4.3.3 and cloud extraction in Section 4.3.4.

Android	Highlights of New Security Features
2.1 (2010)	APK signing v1 (JAR)
4.0 (2011)	Keychain credential storage for non-system apps
4.2 (2012)	Application verification (outside of AOSP)
4.3 (2012)	SELinux logging policies
4.4 (2013)	Verified Boot 1 (warning only) SELinux enforcing (blocking) policies Full-disk encryption
5.0 (2014)	SELinux MAC between system and apps Full-disk encryption by default with TEE-backed storage
6.0 (2015)	Runtime permissions (instead of install-time permissions) Trusted keystore for symmetric keys APK validation (manifest-based)
7.0 (2016)	Stricter DAC file permissions on app storage (751 → 700) Verified Boot 1 (stronly enforced) Keymaster 2 (key attestation & version binding) File-based encryption Uniform system CAs StrictMode file access (removal of `file://`)
8.0 (2017)	Verified Boot 2 (AVB) `seccomp-bpf` system call filtering Project Treble/HIDL abstraction layer Google Play Protect (anti-malware) Keymaster 3 (ID attestation)
9.0 (2018)	APK signing (v3) All "non-privileged" apps run in individual SELinux sandboxes Keymaster 4 (key import) Metadata encryption (things not encrypted by File-based) Stricter seccomp
10.0 (2019)	Limited raw view of FS (no access to `/sdcard`) Removal of full-disk encryption Location access only in foreground Block access to device identifiers Mandatated storage encryption
11.0 (2020)	File-based encryption enhancements Package-visibility filtering Scoped storage enforcement

Table 4.13: History of Android (AOSP) Security Features

This is not an exhaustive list of features. Some Android versions have been omitted.

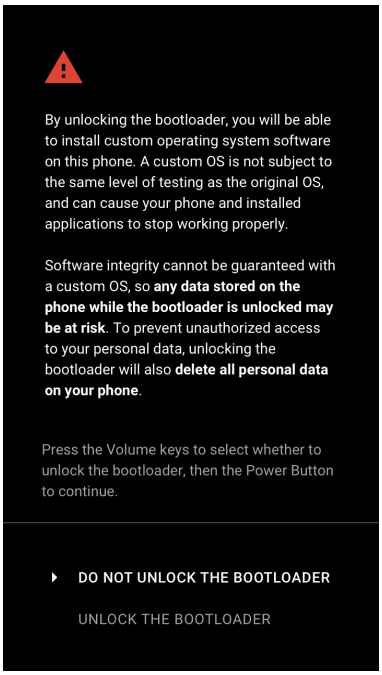

Figure 4.14: Legitimate Bootloader Unlock on Android

Source: Android Open Source Project Documentation [170]

4.3.1 Rooting

Rooting on Android allows a standard user to gain superuser permission on a device, with "a root" being the term for a process that does so successfully. This is analogous to jailbreaking on iOS, as it allows modification of the underlying operating system. Unlike iOS, however, installation of unauthorized (non-Google Play) Android applications is supported (see Section 4.1), making rooting for this reason less likely. Moreover, some devices explicitly support rooting and bootloader modification [229] for non-malicious development purposes [136, 230].

Most roots on Android are related to state of the device bootloader. In locked mode, the bootloader prevents modification to the boot partition, preventing unverified code from running [148, 149], but when in unlocked mode, boot images can be modified, allowing for the application of a root or even the installation of a custom operating system [136, 170, 230]. As a security measure, the process of unlocking the bootloader deletes all personal data on an Android device; the system can no longer trust the bootloader, which means it can no longer assure the data in its storage [148, 230]. Figure 4.14 shows the default message in AOSP when a bootloader is unlocked through this method. Common roots [231, 232] operate under this assumption, and do not attempt to circumvent such a data wipe. However, there are some roots that do not require any kind of bootloader unlock, loading the rooting code via an exploit in the operation system. Some of these roots are publicly available [233], and can be directly sideloaded into an Android device. Other roots must be manually developed for individual devices, which we discuss below.

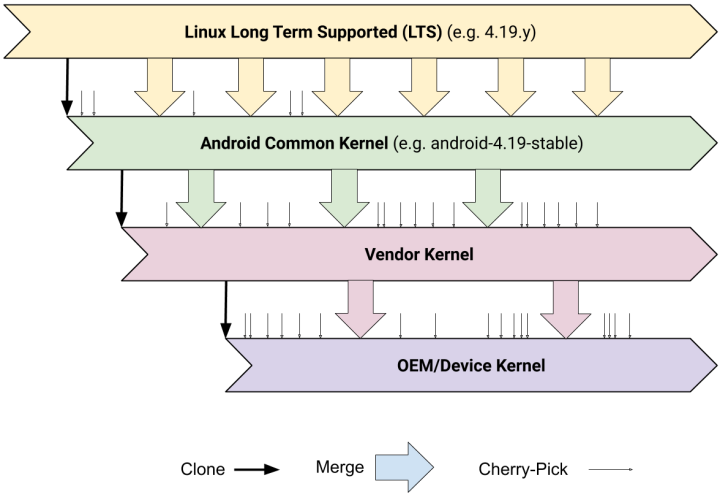

Figure 4.15: Kernel Hierarchy on Android

Source: Android Open Source Project Documentation [221]

Android rooting has been around since the earliest days of the operating system: the first Android device, the HTC Dream (or T-Mobile G1), had a jailbreak found months after its release [234, 235]. At the time, manufacturers and carriers would restrict features (such as USB tethering [236]) behind payment, or install unwanted applications [237]. Users circumvented disabled these features by rooting their devices. However, as manufacturers started to improve software, rooting became less critical [238]. Roots continue to be developed, however [231, 232].

Manufacturers have also moved to reduce the incentives surrounding rooting for non-malicious purposes. Samsung includes the Knox warranty bit, a one-time programmable electronic fuse, in its devices starting in 2013 [239]. This fuse "trips" [240] when it detects a unapproved kernel, and can only be "replaced" with a total teardown of the system board [240]. Moreover, Google's SafetyNet attestation service, which validates if a system can be trusted or not, fails on rooted devices [181]. Passing a SafetyNet check is required for certain services such as Google Pay [183, 184]. Anecdotally [231], Google SafteyNet has been difficult to circumvent. As of 2017, SafetyNet data shows that only 5.6% of active Android devices are rooted (either by consent or maliciously) [194]. Google is currently testing hardware attestation for SafetyNet in some devices, which could further disincentivize legitimate rooting [241],

For the purposes of this discussion, we focus on manual, exploit-based roots. These are roots that are achieved as a result of a vulnerability in Android, and do not trigger an erase of data on the phone, and are potentially more valuable in forensics, as a result. We consider exploitation at two levels: the core operating system (kernel) and device-specific code (SoC vendor and OEM code).

Exploiting the core operating system. An Android device kernel is the combination of

several patches and drivers from various sources, as shown in Figure 4.15. Underlying the entire system is a kernel from the long-term support branch of Linux with Android-specific changes (the Android Common Kernel). On top of this, the system-on-chip vendor can make modifications to better support its hardware, and the final OEM manufacturer of the device can add even more patches [221]. Software vulnerabilities can appear at any of these levels and compromise the entire system. Although Google and others make several patches to the Android kernel, the kernel is still Linux at its core. From a security standpoint, this means that exploits that provide superuser access on Linux may have impact on Android as well. For example, Dirty CoW was a general Linux vulnerability that used a race condition to write to system files [242]; this vulnerability was then applied to Android, and developers created a root exploit based around it [243]. There are also vulnerabilities that were originally discovered on Android that actually impacted the Linux kernel at-large. An example of this is PingPongRoot, a use-after-free vulnerability used in an Android root that required a kernel patch upon disclosure [244].

Zero-day Android root vulnerabilities are still being actively found and exploited [245, 246, 247]. These operating system-level vulnerabilities are attractive for forensics because they are more or less device-agnostic. The fragmented state of Android hardware means that a common operating system exploit can be applied to more devices.

Device-specific bypasses. This class of vulnerabilities is specific to properties associated with a particular device. These vulnerabilities may require more expertise and time to properly exploit, but can completely ignore the protections provided by the Android system. Generally, these bypasses look at a lower level of abstraction, such as system-on-chip firmware.

A high-profile [248, 249] example of this is Qualcomm EDL. Qualcomm devices have an "Emergency Download Mode" (EDL) [250] that allow OEM-signed programmers to flash changes onto the system. When placed into EDL mode (usually by hardware circuit shorts [250]), an Android device boots into a debug interface rather than the usual Android Bootloader. Then, using the tools of a programmer, an attacker can flash data onto the Android partition. These programmers are ostensibly guarded by an OEM, but are often publicly available [250]. By flashing the partition with such a programmer, attackers have been able to attain root and exfiltrate data, even on encrypted devices. For example, EDL can be used to flip the bootloader lock bit on some devices, which can then boot into a custom image with a root shell, which can in turn be used to dump the data from a device [251].

Qualcomm EDL is not the only low-level bypass on Android. Certain MediaTek-based devices can be rewritten (or "flashed") without a bootloader unlock, using the publicly-available SP-Flash-Tool [252]. There is also work on using AT modem commands to circumvent protections on Android [253]. This is not an exhaustive list of all possible bypasses, but a reflection of the security issues that arise due to the complexity and fragmentation in the Android space.

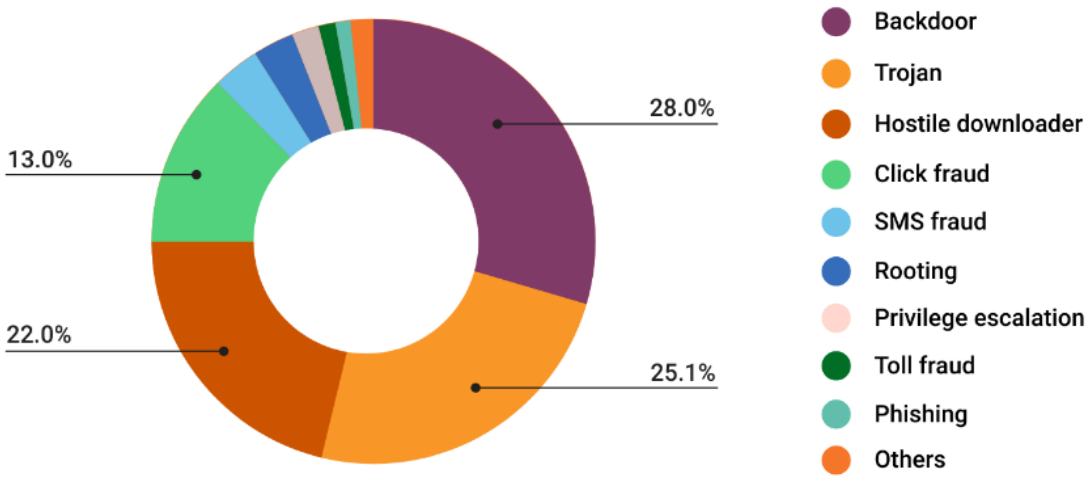

Figure 4.16: Distribution of PHAs for Apps Installed *Outside* of the Google Play Store

Source: Android Year-in-Review 2018 [195]

4.3.2 Alternative Techniques

Root-level access for an Android device, while a sufficient condition to bypass Android permissions checks, is not a necessary one. We briefly discuss alternative methods of bypassing Android security controls that do not strictly rely on rooting the phone as an attack strategy.

Malware and app compromise. Android malware is rare, but has not been eradicated. According to Google-released data, 0.68% of devices that allowed installations from unknown sources had some form of PHA, compared to 0.08% of devices that allowed installation from the Google Play Store exclusively [195]. Google's PHA definition is broad: categories include "click fraud", a form of adware that generates clicks on ads surreptitiously [195] – not a form of PHA that would lead directly to data bypass – as well as rooting apps, even if they were user-wanted. Figure 4.16 is a chart of all PHAs on Android found outside of the Google Play Store. In general, an attacker who has spyware or another form of malware installed already on a device has a clear advantage from a bypass standpoint one that does not. It is non-trivial to have an app installed before forensic analysis, though; an examiner would have to either fool the user into installing a PHA or coerce them to do so.

A second, but related, bypass technique is to leverage insecurities in existing app to bypass protections. An app already on the device may have some kind of open interface or remote code execution vulnerability that can be leveraged to bypass security protections and retrieve information. For example, a double-free vulnerability in WhatsApp could be used to remotely gain a shell and steal from the WhatsApp message database [254]. Compromising an app only allows a limited exploit context, as the regular Android sandboxing protections are still in place. But, it does provide access all of the information of an app and an opportunity to

pivot into the rest of the system.

Trusted hardware attacks. Exploitation of trusted components on Android either target the trusted hardware itself or on the TEEs that run on them. Researchers have found several issues with TrustZone and TrustZone-enabled TEEs over the years [255]. Some TrustZone exploits (as described in [255]) require existing access to the device, in the form of an app installed on the device already. These exploits are less useful for forensics, as the threat model there involves data recovery for systems without prior access. Still, these kinds of attacks could be used as a part of an exploit chain instead, gaining a foothold in the system first, and then attacking TrustZone and the TEE. As of this writing, there are no public security exploits for the Samsung Secure Processor and Google Titan M secure element co-processors, making them valuable targets: Google in particular offers $1,000,000 for a successful exploit of Titan M [226].

Passcode guessing. If an Android device using file-based encryption or full-disk encryption is rebooted, it will request a passcode for decryption. A six-digit PIN or simple pattern can be brute-forced in a manner of minutes on commodity hardware, so Android employs Gatekeeper as a protection mechanism. There are two components of Gatekeeper: a hardware-abstraction layer (HAL), which is an API for clients to enroll and verify passwords, and a trusted application (TA), which actually runs on a TEE. Gatekeeper creates an HMAC for every passcode that it has approved; this HMAC also includes a User Secure Identifier (SID), which is generated by a secure random number generator upon enrollment with Gatekeeper. It validates that HMAC of passcode attempts matches that of the approved passcode. The HMAC key that Gatekeeper uses for this purpose does not leave the TEE. As a brute-force prevention mechanism, Gatekeeper enables request throttling: the TA does not respond to service requests if a timeout is in place [150]. Circumventing Gatekeeper would involve breaking the Gatekeeper TA as implemented in a specific TEE OS, exploiting the entire TEE/TrustZone architecture as discussed above, or finding a loophole that skips the Gatekeeper check entirely. We discuss evidence of such circumvention in Section 4.3.3.

Lock screen bypasses. Lock screen bypasses are cases wherein a user can unlock a phone or gain access to functionality normally restricted while the phone is locked without the user authentication factor or biometric. While Google's Smart Lock feature [154] bypasses the lock screen in some cases, this is by design. We did not find evidence that unauthorized UI-driven lock screen bypasses are commonplace, or relevant to Android forensics.

4.3.3 Local Device Data Extraction

Once an Android device is seized, an examiner can proceed with extracting the data on it. If a device is seized while it was in use, it should have all of its encryption keys in memory already: in full-disk mode [141], the system's user data partitions would already be unlocked, and in file-based mode [138], the credential-encrypted (CE) keys would have to be in use. Of course, in the best case (for law enforcement), the user could have their PIN or password written nearby. With this, a forensics examiner can simply unlock the phone, and use the Android Debug Bridge [207] to pull a backup off the device via USB, obviating the need for

```
10875791+0 records in
10875791+0 records out
5568404992 bytes transferred in 2011.477 secs (2768316 bytes/sec)
root@VF-895N:/ #
```

Figure 4.18: Extracting Data on a Rooted Android Device Using `dd`

Source: Andrea Fortuna [228]

any cracking of the Android device. Also, if the device has an SD card (that is not adopted or encrypted by the system [132]), an examiner can simply pull it out of the system and analyze it directly for data.

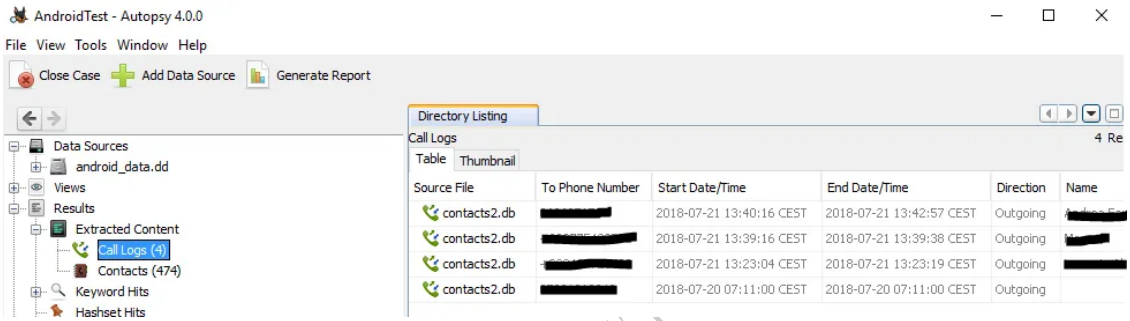

Figure 4.17: Autopsy Forensic Analysis for an Android Disk Image

Source: Andrea Fortuna [228]

If the device is indeed protected, an examiner has several options. The end goal is to extract all of the information off of a device, typically by using some kind of forensic explorer to read organize information. If the device is not secured or has an easily-exploitable vulnerability, (and if the agency has sufficient expertise), they may run exploits directly against the target, and run a tool (such as the open-source Autopsy [256], Figure 4.17) against the device. Many of the roots discussed in Section 4.3.1 were created by members of the Android development community for personal use [231, 232], but can be re-purposed to gain access onto a device and attain a root shell [257]. The shell can then be used to clone the system (using `dd`, Figure 4.18) for analysis off the phone [228].

If additional assistance is required, law enforcement examiners can hire a forensics company, like Cellebrite, Oxygen, or Magnet. These organizations are secretive about their proprietary methodologies, but do leave hints about their methods in marketing documentation [7, 16, 258]. Often, these forensics tools draw from a library of exploits. For example, Cellebrite, Oxygen, and Magnet all employ Qualcomm EDL programmers (described in Section 4.3.1) to extract data [19]. A procedure to put an Alcatel 5044R device in EDL mode for Cellebrite's UFED software can be found in Figure 4.19. EDL programmers must be signed by the vendor, but many of them have been leaked online [250]. It is unknown if forensics organizations are legally allowed to use these binaries for this purpose. Once the bypass is run, forensics

Figure 4.19: Cellebrite EDL Instructions for an Encrypted Alcatel Android Device

Source: Privacy International [19]

software typically analyze and exfiltrate data; example output from Cellebrite UFED is found in Figure 4.20.

4.3.4 Cloud Data Extraction

Android has tight integration with Google services, and as a result Android devices store significant amount of information on Google's servers. GMS apps such as Gmail, Google Drive, and Google Photos store data on the Google Cloud backend with no guarantees of end-to-end encryption. This means that Google has unfettered access to this information, and is able give this information to law enforcement upon request. The only exception to this (according to available information) would be Key/Value Backup data stored in the Android Backup Service [119], which is end-to-end encrypted.

Google complies with law enforcement requests for information, processing 20,000 search warrants for information in the United States in 2019 [259]. Unlike Apple's iCloud, Google's cloud offerings are platform agnostic, and as such Google has information about iPhone users in addition to Android users. Law enforcement has expressed particular interest in Google's store of location data for users [260]. If law enforcement does not wish to go through Google (and potentially have to pay to service their request [261]), forensic software can attempt to recover cloud authentication tokens on a device and then use it to manually download data from an individual's Google account [13, 117].

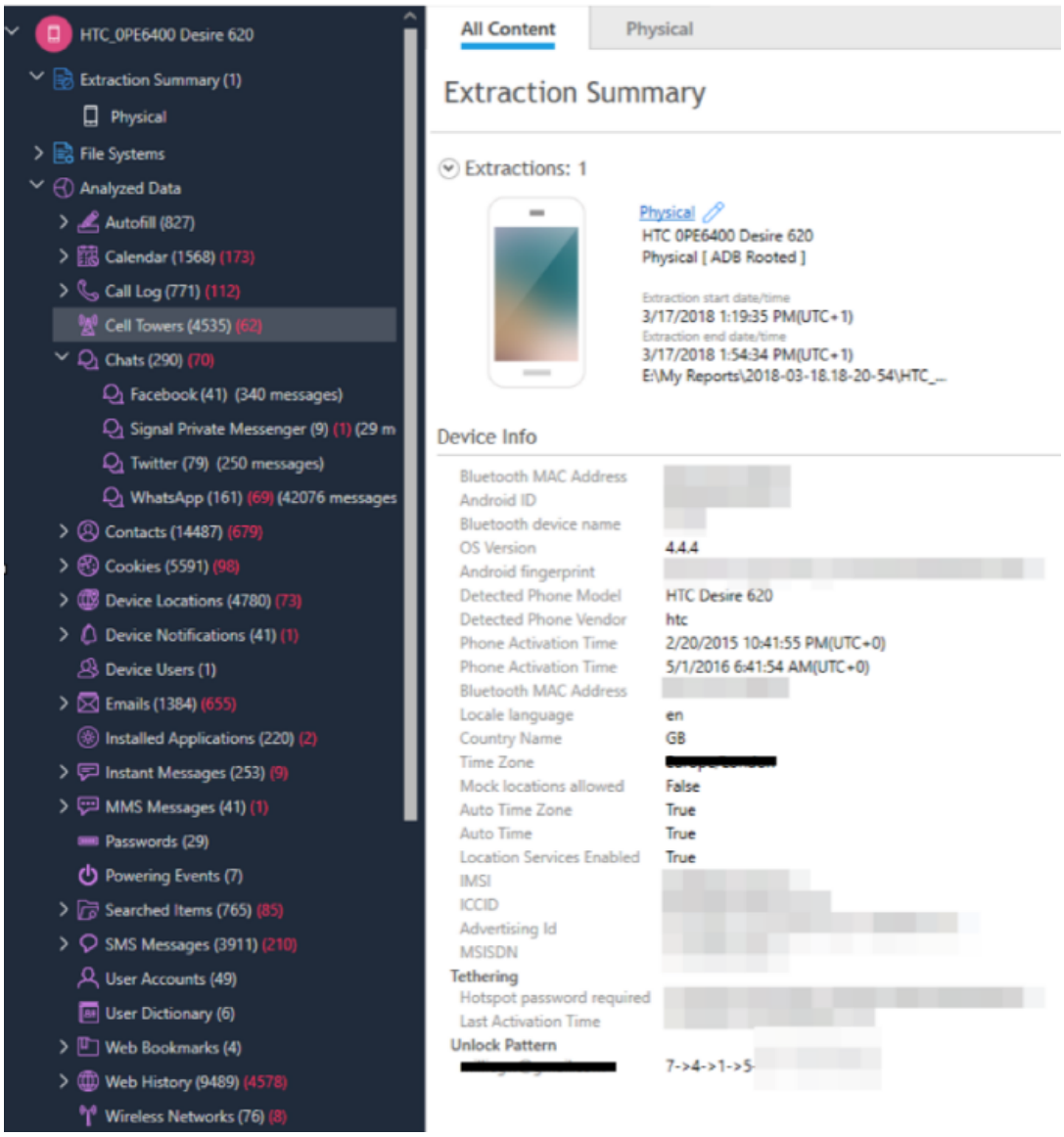

Figure 4.20: Cellebrite UFED Interface During Extraction of an HTC Desire Android Device

Source: Privacy International [19]

4.3.5 Conclusions from Bypasses

The primary takeaway from this discussion is that there are many techniques to bypass user data protections on Android. Lacking an analogue to iOS Complete Protection, decryption keys for user data remain available in memory at all time after the first unlock of the device; live extraction then becomes a question of breaking security controls instead of breaking cryptography or hardware. Additionally, the extent of Google's data collection affords law enforcement and rogue actors alike considerable user data, acquirable either through the legal system or through a device bypass. We discuss further improvements to Android in greater detail in Section 4.5.

4.4 Forensic Software for Android

As Android phones have gained market share, so have Android forensics tools. As with iOS, it is critical to realize the accessibility of professional forensic tools such as Cellebrite's UFED [14], and even of individualized consulting services such as Cellebrite's Advanced Services [7] for unlocking phones. Law enforcement agencies, including local departments, can unlock devices with Advanced Services for as cheap as $2,000 USD per phone, and even less in bulk [17], and commonly do so [17, 40, 19].

For a complete list of the forensic tools tested by DHS, as determined by publicly-available reports as of this writing, and the data forensically extracted by those tools, see Appendix C and archived DHS reports [22]. Again unfortunately, the NIST standard for testing devices is unclear as to whether the device should be in a locked state during testing [42]; we note, however, that certain categories of data seem inaccessible to forensic software in various cases, which we attribute to Android security controls. It is possible that these failures occur due to the forensic tools inability to correctly extract and display data they should have access to, but based on the patterns of inaccessible data in the reports we consider this unlikely. If this assumption holds, in summary these reports imply the following:

Successful extraction. In most tests, most or all of the targeted data (see Figure 2.1) is successfully extracted against the latest Android devices. One notable omission, however, is location data: while forensic tools were largely able to extract data from Android phones, we speculate that since Maps and Location is a GMS component, this data may be stored exclusively in the cloud, leaving none for local extraction.

Limited software diversity. A small number of forensic software companies frequently iterate their products. This is demonstrated in the DHS tests as many Cellebrite, XRY, Oxygen, MOBILedit, Secure View, and Lantern devices being tested between 2009 and 2019 (see Appendix C), and each generally successfully extracting data from contemporaneous generations of Android devices over time [22].

Slow adoption of new test devices. The tests for Android show results for a number of different Android versions, with subsequent tests either remaining on an old version, or skipping over several major versions. Studies would continue to use older Android versions years after they

were released. Even in 2019, the latest version tested in one report was 7.1.1 – a version of Android first tested in 2017 – even though less than 9% of U.S. Android users were on Android 7 at the time [224].

Reporting inconsistencies. Beginning in February 2016, the quality of the reports degrades notably, and it is unclear in many cases whether data was extracted from Android and not displayed properly by the forensic software, or simply not extracted at all. Some reports[4] showed inconsistencies between the analysts notes on forensic performance and the summary tables in the final report (e.g. claimed extracted data in the notes, but this success not indicated in the summary table).

Regardless of irregularities in methodology, it is likely given these reports that agencies such as FBI and DHS that have access to these tools are largely able to bypass Android protections and exfiltrate data from Android devices.

4.5 Proposed Improvements to Android

Android security has improved significantly over the years, with modern Android using more cryptography and security controls than ever before. However, forensics experts are still able to compromise the Android system and bypass its protections. Below, we briefly describe some potential improvements that Google can make to Android to make it more resilient against bypass.

Encrypt user data on screen lock. The most important improvement Android can make would be to encrypt user data with a key that is evicted in memory after the screen is locked. File-based encryption with device and credential-encrypted storage types was a step in the right direction, but adding this protection will ensure cryptographically that personal data is unavailable while the user is not using the device. This is not a trivial change, as it would require modifications to the Android process lifecycle [162] to notify apps that the screen was locked and that their processes no longer have access to their data. This, in turn, would force app developers to adapt their application to the new lifecycle, or risk losing users due to app failures. However, if implemented correctly (and throughout the Android system), such a file encryption mechanism would significantly improve the privacy guarantees of the Android system, and make root exploits less effective.

Implement end-to-end encryption for messaging. Currently, Google rolling out RCS as a part of its Google Messenger app on Android [215]. Google should take the next step, and use its outsize influence to promote end-to-end encryption for RCS. Android currently lacks an end-to-end encrypted messaging solution: not only would end-to-end encrypted RCS improve security for Android users, but tying it to a standard protocol may help increase end-to-end encryption adoption worldwide. This would also allow stronger guarantees about the state of its web Messages application, as end-to-end encryption would tie message security between client and phone directly to cryptography rather than data policy. It appears that Google is preparing for such a development [32].

[4]For example see April 2017 "Electronic Evidence Examiner - Device Seizure v1.0.9466.18457."

Provide end-to-end security for more Google products. The only offering that offers end-to-end security in the Google service ecosystem is Duo for real-time video calls. Google should add end-to-end encryption for more information in its cloud, especially for data generated or stored by GMS apps on Android. This is harder ask for Google, as its services run sophisticated machine learning over user data to improve the quality of its service [262]. However, at the very least, Google should extend its end-to-end encryption guarantees from the Android Backup Service to the Android Auto-Backup service as well. Automatic end-to-end security for users and their app data may create tension with law enforcement agencies [9, 11] but would be a boon for the privacy of users.

Secure the low-level firmware programming interface. Low-level exploits bypass the careful Android security model Google has developed. Firmware programmers are widely available; this makes attacks practical [250, 251], with forensics software [19] employing these methods to bypass security controls. As such, SoC vendors and device OEMs should work on securing access to firm. One option is to distribute programmers as hardware devices with tamper-resistant components. In this way, vendors can keep track of programmers and ensure they are not easily leaked. Another option is to consider revoking the certificates of compromised programmers. Phones would flash their firmware with these revocations, and would not respond to firmware mode if it detects a programmer on its revocation list.

Leverage strong trusted hardware. Secure element co-processors add additional security over shared-processor designs such as TrustZone. More parts of Android should tie itself to a secure element, as the separate hardware is more resistant to hardware and software-bound attacks. Android is a worldwide platform, however, and there may be situations where a secure processor cannot be added to a device for price constraints. TrustZone, while not as robust as a secure element, should be used as a fallback. Recommendations like reducing TEE size and bolstering hardware against side-channel attacks [255] are sensible improvements to improve TrustZone security for devices that do not support a secure element.

Expand update efforts. The latest Android devices seem to employ relatively strong data protection techniques. However, Android phones run older software versions in massive numbers [263]. For devices which are capable of receiving the latest updates, increased efforts for user education and incentivization could improve update numbers, especially paired with improved manufacturer outreach. Devices which have reached end-of-life and cannot receive the latest updates must be clear to users about the risks entailed.

Derive mutual benefit from shared code. As we discuss, Android is based on the Linux kernel, and as such derives both the benefits and drawbacks of such sharing. The open-source Linux kernel is a widely used, deeply analyzed operating system, and Google is a powerful force for vulnerability discovery and mitigation [264]. By dedicating time, resources, and engineering effort toward the betterment of the Linux kernel, Google immediately reaps the benefits of such efforts through improvements to Android. Similarly, Google may benefit immediately from the open-source efforts of improving the security and performance of the Linux kernel.

Embrace the federated nature of Android. One of the greatest strengths of Android is

that it is an open ecosystem of several manufacturers. While this can create fragmentation, this also allows for the opportunity to design secure systems that do not rely on a single party (Google) to function. Google should use Android to champion open protocols and manufacturer interoperability through common standards. This will provide end users with thoughtfully and securely designed systems while still maintaining the level of choice they come to expect from the Android ecosystem.

Chapter 5

Conclusion

Privacy is critical to a free and open society. For one, people behave differently when they know they might be surveilled [265, 266, 267, 268], and so privacy-preserving communication channels can alone enable truly democratic discussion of ideas. As more of our data is gathered and our interactions occur on (particularly mobile) devices, these considerations become more important. Worse, harms from violations of privacy are concentrated among some of the most vulnerable, chronically disenfranchised populations in our societies such as the Uighur population in China [20] and Indigenous protesters in the United States [21].

Smartphones have the potential to fundamentally change the balance of privacy in our lives. Not only do they contain our daily schedules and locations, they store and deliver our communications, rich media content documenting our activities, and are a gateway to the Internet. Our devices also contain information about our families and peers, meaning that compromise of their privacy affects our entire interpersonal network. With the rapid advances of data science, machine learning, and the industry of data aggregation, the potential privacy loss due to phone compromise is difficult to overstate. When we fail to protect our data, we are making privacy decisions not only for ourselves but on behalf of anyone with whom we communicate and interact.

The questions we raise in this work are primarily technical. But they stand for a larger question: how do we improve privacy for users of mobile devices? In this work we demonstrate significant limitations in existing systems, and show the existence of bypasses to security controls which protect our. We also present ideas which we hope can be adopted, extended, and improved towards the overall goal of improving privacy. Policy and legislative solutions are also very relevant toward this end; we leave this analysis for experts in the respective fields [17], but to summarize their work, meaningful opportunities for change include limiting law enforcement rights to search devices, requiring robust data sealing and deletion standards, and increasing transparency through audits and logging.

Many of these limitations are centered around what data is encrypted when, and where encryption keys are stored. Encryption, unlike any operating systems security control, lacks the complex state space which contributes to vulnerabilities in software. Whereas the operating system must contend with and manage various user and system contexts, and correctly provide access control and handling from even potentially malicious sources,

encryption can be summarized in brief: is the data encrypted using a strong cipher, and where are the keys? We find that while much data on iOS and Android is stored encrypted, the keys are often available in memory. This creates an opportunity for a compromised OS kernel to exfiltrate data as we see in various forensic tools and bypasses. Further, in Android we find that many widely-used but outdated versions of Android offer even more limited coverage of encryption, up to as weak as only encrypting data when the device is off. While modern versions offer strong and more granular file-based encryption, older models are relegated to disk encryption; disk encryption is wholly unprepared for the stronger adversaries we consider in our threat model, where running devices may be seized at any time. In the cloud, both platforms extensively store user data on behalf of devices, and while there are options for end-to-end encrypted content such as app developer opt-in backups on Android and certain data categories on iOS, this coverage is limited due to design decisions by Apple and Google.

Secure hardware offers compelling security benefits on both mobile platforms, particularly by giving the devices a place to store encryption keys without risking their immediate compromise. It is these components, whether the Secure Enclave on iOS or TrustZone on Android, which allow mobile devices to contend in our stronger threat model, far beyond what for example most laptop computers could hope to. Secure hardware is the only reliable method of storing encryption keys on-device and protecting them that we find in the industry, modulo potential bypasses of Secure Enclave technology (e.g. by GrayKey) and vulnerabilities in TrustZone.

Going beyond the current state of the art, there are exciting possibilities to create novel methods for increasing the coverage of encryption without limiting performance. This is a deeply technical challenge which may draw in research from cryptography, systems security, and even machine learning in the example of creating prediction systems for limiting available decryption keys. There are also compelling questions about update distribution and synchronization to be answered. Both platforms could benefit greatly from the work of secure multi-party computation researchers in developing privacy-preserving replacements for sensitive but desirable cloud services such as backup, data aggregation, and predictive systems. Additionally, extensive usable security research and other sociological studies can answer questions about how people expect security and privacy controls to work, and compare any gaps with how they are truly implemented. User studies are difficult and expensive, but also have great potential to aid development research by informing them with a human perspective. Follow-up analysis of the effectiveness of security awareness and education works are also critical in this regard.[1]

Privacy lies at the crossroads of technology and policy, of academic and engineering interest, and of producer and consumer effort. Effective privacy controls can be mandated through legislative efforts [270] as well as through technical design considerations [29, 166]. Thus there is much opportunity for improvement, but these intersections complicate solutions due to differences in understanding, language, and motivations amongst policymakers and technology designers. The challenges we highlight in this work, then, can largely be solved through mutually informed efforts in the policy and technical domains. We urge researchers and

[1] A relevant example can be found in [269].

engineers to collaborate with policymakers, advocates, and experts in these efforts. Towards improved usage norms, towards efficient secure protocols to enable sensitive use-cases, and towards mobile device privacy *by default* for all people.

Acknowledgments

The authors would like to thank Upturn, particularly Emma Weil and Harlan Yu, for their work on the recently released document *Mass Extraction* [17] and for their excellent and helpful feedback on this work.

The authors would also like to thank the informal reviewers of this work for their technical contributions.

Bibliography

[1] BankMyCell.com. How Many Smartphones Are In The World? https://www.bankmycell.com/blog/how-many-phones-are-in-the-world, 6 2020. Accessed 2020-07-24.

[2] Pew Research. Mobile Fact Sheet. https://www.pewresearch.org/internet/fact-sheet/mobile/, 6 2019.

[3] Feliks Garcia. iCloud celebrity nude leak: Man pleads guilty to hacking emails of stars including Jennifer Lawrence and Kate Upton. *Independent*, 2016.

[4] Paul Ruggiero and Jon Foote. Cyber Threats to Mobile Phones. US-CERT Publication, Available at https://us-cert.cisa.gov/sites/default/files/publications/cyber_threats_to_mobile_phones.pdf, 2011.

[5] Department of Homeland Security. Study on Mobile Device Security. https://www.dhs.gov/sites/default/files/publications/DHS%20Study%20on%20Mobile%20Device%20Security%20-%20April%202017-FINAL.pdf, 2017.

[6] Vladimir Katalov. The Art of iPhone Acquisition. https://blog.elcomsoft.com/2019/07/the-art-of-iphone-acquisition/, 7 2019. Accessed 2020-08-04.

[7] Cellebrite. Cellebrite Advanced Services, 9 2020. Accessed 2020-09-04.

[8] James Comey. Going Dark: Are Technology, Privacy, and Public Safety on a Collision Course? https://www.fbi.gov/news/speeches/going-dark-are-technology-privacy-and-public-safety-on-a-collision-course, 10 2014. Remarks by former FBI Director James Comey at the Brookings Institution in Washington, D.C. on October 16, 2014, as archived on fbi.gov [Accessed: 2020 07 19].

[9] Answers to your questions about Apple and security. https://www.apple.com/customer-letter/answers/, 2 2016. Accessed 2020-09-22. Date of publication set as earliest snapshot from archive.org at https://web.archive.org/web/20160222150037/https://www.apple.com/customer-letter/answers/.

[10] Apple. A Message to Our Customers. Available at https://www.apple.com/customer-letter/, 2 2016.

[11] James Comey. FBI Director Comments on San Bernardino Matter. https://www.fbi.gov/news/pressrel/press-releases/fbi-director-comments-on-san-bernardino-matter, 2 2016. Accessed 2020-09-22.

[12] A Special Inquiry Regarding the Accuracy of FBI Statements Concerning its Capabilities to Exploit an iPhone Seized During the San Bernardino Terror Attack Investigation. https://www.oversight.gov/sites/default/files/oig-reports/o1803.pdf, 3 2018. Accessed 2020-09-22. Mirrored for archival.

[13] Cellebrite. Unlock cloud-based evidence to solve the case sooner. https://www.cellebrite.com/en/ufed-cloud/, 9 2020. Accessed 2020-09-10.

[14] Cellebrite. What Happens When You Press that Button? https://smarterforensics.com/wp-content/uploads/2014/06/Explaining-Cellebrite-UFED-Data-Extraction-Processes-final.pdf, 6 2014. Accessed 2020-09-26. Mirrored for archival.

[15] Elcomsoft. iOS Forensic Toolkit 6.50: jailbreak-free extraction without an Apple Developer Account. https://www.elcomsoft.com/news/762.html, 9 2020. Accessed 2020-09-22.

[16] Oxygen Forensics. Oxygen Forensic® Detective v.12.6. https://blog.oxygen-forensic.com/oxygen-forensic-detective-v-12-6/, 7 2020. Accessed 2020-09-04.

[17] Logan Koepke, Emma Weil, Urmila Janardan, Tinuola Dada, and Harlan Yu. Mass Extraction: The Widespread Power of U.S. Law Enforcement to Search Mobile Phones. https://www.upturn.org/reports/2020/mass-extraction/, 10 2020. Accessed 2020-10-25. Mirrored for archival.

[18] Joseph Menn. Exclusive: Apple dropped plan for encrypting backups after FBI complained - sources. *Reuters*, 1 2020. Accessed 2020-09-13.

[19] Privacy International. A technical look at Phone Extraction. https://privacyinternational.org/sites/default/files/2019-10/A%20technical%20look%20at%20Phone%20Extraction%20FINAL.pdf, 10 2019. Accessed 2020-09-22. Mirrored for archival.

[20] Xiao Qiang. The road to digital unfreedom: President xi's surveillance state. *Journal of Democracy*, 30(1):53–67, 2019.

[21] Jenna Harb and Kathryn Henne. Disinformation and resistance in the surveillance of indigenous protesters. In *Information, Technology and Control in a Changing World*, pages 187–211. Springer, 2019.

[22] United States Department of Homeland Security. Test Results for Mobile Device Acquisition. https://www.dhs.gov/publication/st-mobile-device-acquisition, 10 2019. Accessed 2020-08-04, reports mirrored for archival.

[23] Peter Cao. This is the 'GrayKey' box used by law enforcement to unlock iPhones [Gallery]. https://9to5mac.com/2018/03/15/this-is-the-graykey-box-used-by-law-enforcement-to-unlock-iphones-gallery/, 3 2018. Accessed 2020-09-10.

[24] S.3398 - EARN IT Act of 2020. https://www.congress.gov/bill/116th-congress/senate-bill/3398, 3 2020. Senate bill introduced by Sen. Lindsey Graham. Accessed 2020-09-22.

[25] Thomas Brewster. Apple Helps FBI Track Down George Floyd Protester Accused Of Firebombing Cop Cars. https://www.forbes.com/sites/thomasbrewster/2020/09/16/apple-helps-fbi-track-down-george-floyd-protester-accused-of-firebombing-cop-cars/, 9 2020. Accessed 2020-09-21.

[26] Apple Inc. Transparency Report. https://www.apple.com/legal/transparency/, 9 2020. Accessed 2020-09-21.

[27] Apple Inc. iOS Security. TODOgithub, 2012–2019. iOS Security Guides. Mirrored for archival.

[28] Apple Inc. Apple Platform Security. TODOgithub, 2019–2020. Mirrored for archival.

[29] Apple Inc. Privacy. https://www.apple.com/privacy/, 9 2020. Accessed 2020-09-25.

[30] Untitled. https://assets.documentcloud.org/documents/20393462/gkey-tracker-responses_redacted.pdf. Accessed 2020-11-07.

[31] Apple Inc. iCloud Security Overview. https://support.apple.com/en-us/HT202303, 7 2020. Accessed 2020-07-28.

[32] Kim Lyons. Google Messages may finally be adding end-to-end encryption for RCS . https://www.theverge.com/2020/5/23/21268577/google-messages-end-to-end-encryption-rcs, 5 2020. Accessed 2020-09-25.

[33] S. O'Dea. Manufacturers' market share of smartphone subscribers in the United States from 2013 and 2019, by month. https://www.statista.com/statistics/273697/market-share-held-by-the-leading-smartphone-manufacturers-oem-in-the-us/, 11 2019. Accessed 2020-09-25.

[34] Android. Android Certified Partners. https://www.android.com/certified/partners/, 2020. Accessed 2020-09-25.

[35] ARM Holdings. Arm TrustZone Technology. https://developer.arm.com/ip-products/security-ip/trustzone, 9 2020. Accessed 2020-09-09.

[36] Sergei Skorobogatov. The bumpy road towards iphone 5c nand mirroring. *arXiv preprint arXiv:1609.04327*, 2016.

[37] Russell Brandom. A new hack could let thieves bypass the iPhone's lockscreen. https://www.theverge.com/2015/3/30/8311835/iphone-lockscreen-hack-theft-find-my-iphone, 3 2015. Accessed 2020-09-09.

[38] Joseph Cox and Izzie Ramirez. iPhone Warrant Database 2019. https://docs.google.com/spreadsheets/d/1Xmh1QEXYJmVPFlqAdEIVGemvbkoZmk_WyAPGC4eY-eE/edit#gid=0, 3 2020. Mirrored for archival.

[39] Russell Brandom. Police are filing warrants for Android's vast store of location data. https://www.theverge.com/2016/6/1/11824118/google-android-location-data-police-warrants, 6 2016. Accessed 2020-09-25.

[40] Joseph Cox. We Built a Database of Over 500 iPhones Cops Have Tried to Unlock. https://www.vice.com/en_us/article/4ag5yj/unlock-apple-iphone-database-for-police, 3 2020. Accessed 2020-09-22.

[41] Patrick Siewert. Apple iPhone Forensics: Significant Locations. https://www.forensicfocus.com/articles/apple-iphone-forensics-significant-locations/, 5 2018. Accessed 2020-09-22.

[42] United States National Institute of Standards and Technology. Mobile Device Forensic Tool Specification, Test Assertions and Test Cases. https://www.nist.gov/system/files/documents/2019/07/11/mobile_device_forensic_tool_test_spec_v_3.0.pdf, 5 2019. Accessed 2020-08-04, mirrored for archival.

[43] Ryan Browne. Apple becomes biggest smartphone player for the first time in two years, beating Samsung. https://www.cnbc.com/2020/01/30/apple-iphone-beats-samsung-for-smartphone-shipments-in-q4-2019.html, 2020. Accessed 18-July-2020.

[44] Daphne Leprince-Ringuet. Tablet sales are still in decline, but Apple's latest iPad is showing the way forward. https://www.zdnet.com/article/tablet-sales-are-still-in-decline-but-apples-latest-ipad-is-showing-the-way-forward/, 2 2020. Accessed 2020-07-20.

[45] S. O'Dea. Manufacturers' market share of smartphone subscribers in the United States from 2013 and 2019, by month. https://www.statista.com/statistics/273697/market-share-held-by-the-leading-smartphone-manufacturers-oem-in-the-us/, 6 2020. Accessed 2020-09-22.

[46] Dami Lee. Apple says there are 1.4 billion active Apple devices. https://www.theverge.com/2019/1/29/18202736/apple-devices-ios-earnings-q1-2019, 1 2019. Accessed 2020-07-21.

[47] Apple Inc. Example Payouts. https://developer.apple.com/security-bounty/payouts/, 7 2020. Accessed 2020-07-21.

[48] Zerodium. Our Exploit Acquisition Program. https://zerodium.com/program.html, 7 2020. Accessed 2020-07-21.

[49] John Snow. Pegasus: The ultimate spyware for iOS and Android. https://www.kaspersky.com/blog/pegasus-spyware/14604/, 4 2017. Accessed 2020-07-21 from Kaspersky.

[50] Apple Inc. ios11-iphone7-setup-touch-id-passcode.jpg. https://support.apple.com/library/content/dam/edam/applecare/images/en_US/iOS/ios11-iphone7-setup-touch-id-passcode.jpg, 9 2020. Accessed 2020-09-16.

[51] Philipp Markert, Daniel V Bailey, Maximilian Golla, Markus Dürmuth, and Adam J AviG. This pin can be easily guessed: Analyzing the security of smartphone unlock pins. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 286–303. IEEE, 2020.

[52] Apple Inc. Enable iCloud key-value storage, document storage, or CloudKit. https://help.apple.com/xcode/mac/current/#/dev52452c426, 7 2020. Accessed 2020-07-28.

[53] Apple Developer Enterprise Program. https://developer.apple.com/programs/enterprise/, 9 2020. Accessed 2020-09-07.

[54] Tielei Wang, Hao Xu, and Xiaobo Chen. Pangu 9 Internals. https://papers.put.as/papers/ios/2016/us-16-Pangu9-Internals.pdf, 8 2016. Accessed 2020-08-11.

[55] Britta Gustafson. Misuse of enterprise and developer certificates. https://www.theiphonewiki.com/wiki/Misuse_of_enterprise_and_developer_certificates, 8 2016. Accessed 2020-09-04. Multiple authors provided edits.

[56] Tao Wei, Min Zheng, Hui Xue, and Dawn Song. Apple without a shell ios under targeted attack. In *Virus Bulletin Conference*, 2014.

[57] Apple Inc. application_code_signing_2x.png. https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Art/application_code_signing_2x.png, 4 2018. Accessed 2020-09-16.

[58] Apple Inc. App Review. https://developer.apple.com/app-store/review/, 2020. Accessed 2020-07-19.

[59] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, pages 177–183, 2011.

[60] Apple Inc. iOS 14. https://www.apple.com/ios/ios-14/, 9 2020. Accessed 2020-09-16.

[61] Bingkun Guo. fig2.png. https://www.cse.wustl.edu/~jain/cse571-14/ftp/ios_security/index.html, 12 2014. Accessed 2020-09-16.

[62] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In *International Conference on Smart Card Research and Advanced Applications*, pages 277–284. Springer, 1998.

[63] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[64] Burt Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. Technical report, RFC 2898, september, 2000.

[65] Apple Inc. What does iCloud back up? https://support.apple.com/en-us/HT207428, 9 2020. Accessed 2020-09-09.

[66] Apple Inc. Technical Q&A QA1719. https://developer.apple.com/library/archive/qa/qa1719/_index.html, 5 2016. Accessed 2020-09-15.

[67] Andy Greenberg. The Clever Cryptography Behind Apple's 'Find My' Feature. https://www.wired.com/story/apple-find-my-cryptography-bluetooth/, 6 2019. Accessed 2020-07-19.

[68] Apple Inc. Our smartest keyboard ever. https://www.apple.com/my/ios/whats-new/quicktype/, 9 2020. Accessed 2020-09-15.

[69] CloudKit end-to-end encryption. https://support.apple.com/guide/security/cloudkit-end-to-end-encryption-sec3cac31735/1/web/1, 9 2020. Accessed 2020-09-07.

[70] Apple Inc. Two-factor authentication for Apple ID. https://support.apple.com/en-us/HT204915, 7 2020. Accessed 2020-07-28.

[71] Thomas D Wu et al. The secure remote password protocol. In *NDSS*, volume 98, pages 97–111. Citeseer, 1998.

[72] Ivan Krstic. Behind the Scenes with iOS Security. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf, 8 2016. Accessed 2020-09-07.

[73] Apple Inc. FaceID Security. TODOgithub, 11 2017. Mirrored for archival.

[74] Tarjei Mandt, Mathew Solnik, and David Wang. Demystifying the secure enclave processor. *Black Hat Las Vegas*, 2016.

[75] iFixit. iPhone X Teardown. https://www.ifixit.com/Teardown/iPhone+X+Teardown/98975, 11 2017. Accessed 2020-07-21.

[76] iFixit. iPhone 5s Teardown. https://www.ifixit.com/Teardown/iPhone+5s+Teardown/17383, 9 2013. Accessed 2020-07-21.

[77] Apple Inc. Apple Pay security and privacy overview. https://support.apple.com/en-us/HT203027, 7 2020. Accessed 2020-07-30.

[78] Apple Inc. Accessing Keychain Items with Face ID or Touch ID. https://developer.apple.com/documentation/localauthentication/accessing_keychain_items_with_face_id_or_touch_id, 9 2020. Accessed 2020-09-2016.

[79] Oleg Afonin. This $39 Device Can Defeat iOS USB Restricted Mode. https://blog.elcomsoft.com/2018/07/this-9-device-can-defeat-ios-usb-restricted-mode/, 7 2018. Accessed 2020-09-23.

[80] Vladimir Katalov. Working Around the iPhone USB Restricted Mode. https://blog.elcomsoft.com/2020/05/iphone-usb-restricted-mode-workaround/, 5 2020. Accessed 2020-11-07.

[81] Mihir Bellare and Igors Stepanovs. Security under message-derived keys: signcryption in imessage. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 507–537. Springer, 2020.

[82] Mark Baugher, D McGrew, M Naslund, E Carrara, and Karl Norrman. The secure real-time transport protocol (srtp), 2004.

[83] Apple Inc. iOS and iPadOS Usage. https://developer.apple.com/support/app-store/, 6 2020. Accessed 2020-09-10.

[84] Apple Inc. Apple Security Updates. https://support.apple.com/en-us/HT201222, 2003–2020. Accessed 2020-06 through 2020-07.

[85] Apple Inc. Using USB accessories with iOS 11.4.1 and later. https://support.apple.com/en-us/HT208857, 8 2020. Accessed 2020-08-04.

[86] iFixit. A Peek Inside Apple's A4 Processor. https://www.webcitation.org/6DGM9y39v?url=http://www.ifixit.com/blog/2010/04/05/a-peek-inside-apples-a4-processor/, 4 2010. Archived on 2012-12-29, accessed on 2020-07-19.

[87] iFixit. iPad Mini Wi-Fi Teardown. https://www.ifixit.com/Teardown/iPad+Mini+Wi-Fi+Teardown/11423, 11 2012. Accessed on 2020-07-19.

[88] iFixit. Apple A6 Teardown. https://www.ifixit.com/Teardown/Apple+A6+Teardown/10528, 9 2012. Accessed on 2020-07-19.

[89] iFixit. iPhone 6 Teardown. https://www.ifixit.com/Teardown/iPhone+6+Teardown/29213, 9 2014. Accessed 2020-09-15.

[90] iFixit. iPhone 6S Teardown. https://www.ifixit.com/Teardown/iPhone+6s+Teardown/48170, 9 2015. Accessed 2020-09-15.

[91] iFixit. iPhone 7 Teardown. https://www.ifixit.com/Teardown/iPhone+7+Teardown/67382, 9 2016. Accessed 2020-09-15.

[92] iFixit. iPhone XS Teardown. https://www.ifixit.com/Teardown/iPhone+XS+and+XS+Max+Teardown/113021, 9 2018. Accessed 2020-09-15.

[93] iFixit. iPhone 11 Pro Max Teardown. https://www.ifixit.com/Teardown/iPhone+11+Pro+Max+Teardown/126000, 9 2019. Accessed 2020-07-27.

[94] Matt Drange. FBI Hacks Into San Bernardino Shooter's iPhone Without Apple's Help, Drops Case. https://www.forbes.com/sites/mattdrange/2016/03/28/fbi-gets-into-san-bernardino-iphone-without-apples-help-court-vacates-order/, 3 2016. Accessed 2020-09-21.

[95] Apple Inc. Legal Process Guidelines. https://www.apple.com/legal/privacy/law-enforcement-guidelines-us.pdf, 12 2018. Accessed 2020-09-21. Mirrored for archival.

[96] Jay Freeman. Welcome to Cydia. https://cydia.saurik.com/, 2 2008. Accessed 2020-08-11.

[97] Oleg Afonin. Extracting and Decrypting iOS Keychain: Physical, Logical and Cloud Options Explored. https://blog.elcomsoft.com/2020/08/extracting-and-decrypting-ios-keychain-physical-logical-and-cloud-options-explored/, 8 2020. Accessed 2020-09-04.

[98] Michael SW Lee and Ian Soon. Taking a bite out of apple: Jailbreaking and the confluence of brand loyalty, consumer resistance and the co-creation of value. *Journal of Product & Brand Management*, 2017.

[99] Josh Lowensohn. Apple's 2013 by the numbers: 150M iPhones, 71M iPads. https://www.cnet.com/news/apples-2013-by-the-numbers-150m-iphones-71m-ipads/, 10 2013. Accessed 2020-08-11.

[100] MiniCreo. How Many iPhones Are There. https://www.minicreo.com/news/how-many-iphones-are-there.html, 11 2019. Accessed 2020-08-12.

[101] Technical analysis of the checkm8 exploit.

[102] Oleg Afonin. Protecting Your Data and Apple Account If They Know Your iPhone Passcode. https://blog.elcomsoft.com/2018/06/protecting-your-data-and-apple-account-if-they-know-your-iphone-passcode/, 6 2018. Accessed 2020-09-22.

[103] Thomas Reed. GrayKey iPhone unlocker poses serious security concerns. *MalwareBytes SecurityWorld*, 3 2018. Accessed 2020-09-19.

[104] Thomas Brewster. The FBI Got Data From A Locked iPhone 11 Pro Max—So Why Is It Demanding Apple Unlock Older Phones? https://www.forbes.com/sites/thomasbrewster/2020/01/15/the-fbi-got-data-from-a-locked-iphone-11-pro-max--so-why-is-it-demanding-apple-unlock-older-phones/, 1 2020. Accessed 2020-09-24.

[105] Michael Reigle. Baris Ali Koch iPhone 11 Pro Max Search. https://www.documentcloud.org/documents/6656969-Baris-Ali-Koch-iPhone-11-Pro-Max-Search.html, 10 2019. Warrant archived by DocumentCloud. Accessed 2020-09-22.

[106] Robert Palazzo. FCC ID 2AV7EGK01. https://fccid.io/2AV7EGK01, 7 2020. Published by the FCC, accessed via unofficial viewer. Images mirrored for archival.

[107] Lorenzo Franceschi-Bicchierai and Joseph Cox. Here Are Detailed Photos of iPhone Unlocking Tech GrayKey. https://www.vice.com/en_us/article/v7gkpx/graykey-grayshift-photos-iphone-unlocking-tech, 9 2020. Accessed 2020-09-20.

[108] Heiko Ehrenberg. Ieee standard for test access port and boundary-scan architecture. *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pages 1–444, 2013. Working Group chair listed as author. Accessed from https://standards.ieee.org/standard/1149_1-2013.html.

[109] The Phone Wiki. purplera1n. https://www.theiphonewiki.com/wiki/Purplera1n, 3 2017. Accessed 2020-07-29. The Phone Wiki was created by George Hotz (geohot).

[110] Sébastien Page. iPhone 3GS Jailbreak PurpleRa1n Gets An Update. https://www.idownloadblog.com/2009/07/04/iphone-3gs-jailbreak-purplera1n-update/, 7 2009. Accessed 2020-07-29.

[111] Lewis Leong. Chinese developers release untethered iOS 7.1.X jailbreak to much controversy. https://en.softonic.com/articles/pangu-ios-7-1-x-jailbreak, 6 2014. Accessed 2020-07-29.

[112] Raul Siles. Bypassing iOS Lock Screens: A Comprehensive Arsenal of Vulns. https://blog.dinosec.com/2014/09/bypassing-ios-lock-screens.html, 6 2020. Accessed 2020-08-11.

[113] Richard Ayers, Sam Brothers, and Wayne Jansen. Guidelines on Mobile Device Forensics. https://csrc.nist.gov/publications/detail/sp/800-101/rev-1/final, 5 2014. Accessed 2020-08-04, mirrored for archival.

[114] Bob Lee. Detecting Location Using WIFI Hotspots. https://patentimages.storage.googleapis.com/e2/bb/50/51db61bf033480/US9113344.pdf, 8 2015.

[115] Apple Inc. Location Services & Privacy. https://support.apple.com/en-us/HT207056, 9 2019. Accessed 2020-09-22.

[116] Oleg Alfonin. Accessing iCloud With and Without a Password in 2019. https://blog.elcomsoft.com/2019/07/accessing-icloud-with-and-without-a-password-in-2019/, 7 2019. Accessed 2020-09-10.

[117] Privacy International. Cloud extraction technology: the secret tech that lets government agencies collect masses of data from your apps. https://privacyinternational.org/long-read/3300/cloud-extraction-technology-secret-tech-lets-government-agencies-collect-masses-data, 1 2020. Accessed 2020-08-21.

[118] Richard P Ayers, Wayne Jansen, Nicolas Cilleros, and Ronan Daniellou. Cell phone forensics tools: An overview and analysis. Technical report, U.S National Institute of Standards and Technology, 2005. Interagency/Internal Report (NISTIR)-7250.

[119] Troy Kensinger. Google and Android have your back by protecting your backups. https://security.googleblog.com/2018/10/google-and-android-have-your-back-by.html, 10 2018. Accessed 2020-09-20.

[120] Oleg Afonin. iOS 11 Horror Story: the Rise and Fall of iOS Security. https://blog.elcomsoft.com/2017/11/ios-11-horror-story-the-rise-and-fall-of-ios-security/, 11 2017. Accessed 2020-09-24.

[121] Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. Certificate Transparency Version 2.0. Internet-Draft draft-ietf-trans-rfc6962-bis-34, Internet Engineering Task Force, 11 2019. Work in Progress.

[122] Moxie Marlinspike and Trevor Perrin. Trust assertions for certificate keys. http://tack.io/draft.html, 1 2013. Internet Draft. Accessed 2020-11-05.

[123] Design Justice Network. Design Justice Network Principles. https://designjustice.org/read-the-principles, 8 2018. Accessed 2020-09-26.

[124] Sasha Costanza-Chock. Design justice: towards an intersectional feminist framework for design theory and practice. *Proceedings of the Design Research Society*, 2018.

[125] Apple Inc. Apple security research device program. https://developer.apple.com/programs/security-research-device/, 08 2020.

[126] S. O'Dea. Mobile operating systems' market share worldwide from January 2012 to July 2020. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, 8 2020. Accessed 2020-09-22.

[127] Tarun Pathak. Nokia Leads the Global Rankings in Updating Smartphone Software and Security. https://www.counterpointresearch.com/nokia-leads-global-rankings-updating-smartphone-software-security/, 8 2019. Accessed 2020-09-09.

[128] Open Handset Alliance. At https://www.openhandsetalliance.com/.

[129] Google. Google Mobile Services. https://www.android.com/gms/, 2020.

[130] Android Developers. Google Play Store. https://developer.android.com/distribute/google-play/, 2020.

[131] Amazon. Fire OS Overview. https://developer.amazon.com/docs/fire-tv/fire-os-overview.html, 2020.

[132] Android Open Source Project. Adoptable Storage. https://source.android.com/devices/storage/adoptable, 9 2020. Accessed 2020-09-18.

[133] Android Open Source Project. Application Sandbox. https://source.android.com/security/app-sandbox, 9 2020. Accessed 2020-09-09.

[134] Android Open Source Project. Application Signing. https://source.android.com/security/apksigning, 9 2020. Accessed 2020-09-09.

[135] Android Open Source Project. Authentication. https://source.android.com/security/authentication, 9 2020. Accessed 2020-09-09.

[136] Android Open Source Project. Device State. https://source.android.com/security/verifiedboot/device-state, 9 2020. Accessed 2020-09-09.

[137] Android Open Source Project. Implementing dm-verity. https://source.android.com/security/verifiedboot/dm-verity, 9 2020. Accessed 2020-09-09.

[138] Android Open Source Project. File-Based Encryption. https://source.android.com/security/encryption/file-based, 9 2020. Accessed 2020-09-09.

[139] Android Open Source Project. Fingerprint HIDL. https://source.android.com/security/authentication/fingerprint-hal, 9 2020. Accessed 2020-09-09.

[140] Android Open Source Project. fs-verity Integration. https://source.android.com/security/features/apk-verity, 9 2020. Accessed 2020-09-18.

[141] Android Open Source Project. Full-Disk Encryption. https://source.android.com/security/encryption/full-disk, 9 2020. Accessed 2020-09-09.

[142] Android Open Source Project. Hardware-based Keystore. https://source.android.com/security/keystore, 9 2020. Accessed 2020-09-09.

[143] Android Open Source Project. Metadata Encryption. https://source.android.com/security/encryption/metadata, 9 2020. Accessed 2020-09-09.

[144] Android Open Source Project. Android 9 Release Notes. https://source.android.com/setup/start/p-release-notes, 9 2020. Accessed 2020-09-09.

[145] Android Open Source Project. Scoped Storage. https://source.android.com/devices/storage/scoped, 9 2020. Accessed 2020-09-09.

[146] Android Open Source Project. Security-Enhanced Linux in Android. https://source.android.com/security/selinux, 9 2020. Accessed 2020-09-09, mirrored for archival.

[147] Android Open Source Project. Trusty TEE. https://source.android.com/security/trusty, 9 2020. Accessed 2020-09-09.

[148] Android Open Source Project. Verified Boot. https://source.android.com/security/verifiedboot, 9 2020. Accessed 2020-09-09.

[149] Android Open Source Project. Verifying Boot. https://source.android.com/security/verifiedboot/verified-boot, 9 2020. Accessed 2020-09-09.

[150] Android Open Source Project. Gatekeeper. https://source.android.com/security/authentication/gatekeeper, 9 2020.

[151] Adam J Aviv, Katherine L Gibson, Evan Mossop, Matt Blaze, and Jonathan M Smith. Smudge attacks on smartphone touch screens. *Woot*, 10:1–7, 2010.

[152] Adam J Aviv, Devon Budzitowski, and Ravi Kuber. Is bigger better? comparing user-generated passwords on 3x3 vs. 4x4 grid sizes for android's pattern unlock. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 301–310, 2015.

[153] Android Open Source Project. Face Authentication HIDL. https://source.android.com/security/biometric/face-authentication, 9 2020.

[154] Android Help. Choose when your Android phone can stay unlocked. https://support.google.com/android/answer/9075927, 2020. Accessed 2020-09-18.

[155] Rita El Khoury. Trusted Face smart unlock method has been removed from Android devices. https://www.androidpolice.com/2019/09/04/trusted-face-smart-unlock-method-has-been-removed-from-android-devices/, 9 2019. Accessed 2020-09-25.

[156] Richard Chirgwin. Full frontal vulnerability: Photos can still trick, unlock Android mobes via facial recognition. https://www.theregister.com/2019/01/04/photos_trick_smartphones/, 1 2019. Accessed 2020-09-25.

[157] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Ndss*, volume 310, pages 20–38, 2013.

[158] Android Open Source Project. SELinux Concepts. https://source.android.com/security/selinux/concepts, 9 2020. Accessed 2020-09-22.

[159] Milan Broz. DMCrypt. https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt, 08 2020. Accessed 2020-09-22.

[160] Linux Kernel Developers. Filesystem-level encryption (fscrypt). https://www.kernel.org/doc/html/latest/filesystems/fscrypt.html. Accessed 2020-09-22.

[161] Android Developers. Support Direct Boot mode. https://developer.android.com/training/articles/direct-boot, 12 2019. Accessed 2020-09-09.

[162] Android Developers. Behavior changes: all apps. https://developer.android.com/guide/components/activities/process-lifecycle, 3 2020. Accessed 2020-09-23.

[163] Robert Triggs. Arm vs x86: Instruction sets, architecture, and all key differences explained. https://www.androidauthority.com/arm-vs-x86-key-differences-explained-568718/, 6 2020. Accessed 2020-09-25.

[164] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.

[165] Samsung Newsroom. Strengthening Hardware Security with Galaxy S20's Secure Processor. `https://news.samsung.com/global/strengthening-hardware-security-with-galaxy-s20s-secure-processor`, 5 2020. Accessed 2020-09-09.

[166] Xiaowen Xin. Titan M makes Pixel 3 our most secure phone yet. `https://www.blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/`, 10 2018. Accessed 2020-09-09.

[167] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Prime+ count: Novel cross-world covert channels on arm trustzone. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 441–452, 2018.

[168] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptol. ePrint Arch.*, 2016:980, 2016.

[169] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.

[170] Android Open Source Project. Boot flow. `https://source.android.com/security/verifiedboot/boot-flow`, 9 2020. Accessed 2020-09-23.

[171] Ron Amadeo. Google's iron grip on Android: Controlling open source by any means necessary. `https://arstechnica.com/gadgets/2018/07/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/`, 7 2018. Accessed 2020-09-25.

[172] Google Play Help. See your Play Protect certification status. `https://support.google.com/googleplay/answer/7165974`, 2020.

[173] Paul Sawers. By calling Huawei Android fork a security risk, Google contradicts its own open source arguments. `https://venturebeat.com/2019/06/07/by-calling-huawei-android-fork-a-security-risk-google-contradicts-its-own-open-source-arguments/`, 6 2019. Accessed 2020-09-25.

[174] Google Cloud. How G Suite uses encryption to protect your data. `TODOgithub`, 8 2020. Accessed 2020-09-25, mirrored for archival.

[175] Google Cloud Help. Security. `https://support.google.com/android/answer/9075927`, 2020. Accessed 2020-09-18.

[176] Google Duo Help. Your calls stay private with end-to-end encryption. `https://support.google.com/android/answer/2819582`, 2020.

[177] Emad Omara. Google Duo End-to-End Encryption Overview. `TODOgithub`, 6 2020.

[178] Google. Overview of Google Play Services. `https://developers.google.com/android/guides/overview`, 7 2020.

[179] Google. Set up Google Play Services. `https://developers.google.com/android/guides/setup`, 9 2020.

[180] Google. SafetyNet. `https://developers.google.com/android/reference/com/google/android/gms/safetynet/SafetyNet`, 9 2017.

[181] Android Developers. SafetyNet Attestation API. `https://developer.android.com/training/safetynet/attestation`, 8 2020.

[182] Rita El Khoury. Netflix was just the start: Google Play Console lets developers exclude app availability for devices that don't pass SafetyNet. https://www.androidpolice.com/2017/05/18/netflix-just-start-google-play-console-lets-developers-exclude-app-availability-devices-dont-pass-safetynet/, 5 2017. Accessed 2020-09-25.

[183] Mishaal Rahman. Google Pay tests showing SafetyNet status on the home page and protecting Online Purchases with a PIN. https://www.xda-developers.com/google-pay-tests-safetynet-status-protecting-online-purchases-pin/, 10 2019.

[184] Kyle Bradshaw. Google Pay tests showing SafetyNet status on the home page and protecting Online Purchases with a PIN. https://9to5google.com/2020/04/24/google-pay-works-android-11-dp3/, 4 2020.

[185] Android Open Source Project. APK Signature Scheme v2. https://source.android.com/security/apksigning/v2, 9 2020. Accessed 2020-09-23.

[186] Android Developers. Sign your app. https://developer.android.com/studio/publish/app-signing, 8 2020. Accessed 2020-09-09.

[187] Linux Kernel Developers. fs-verity: read-only file-based authenticity protection. https://www.kernel.org/doc/html/latest/filesystems/fsverity.html. Accessed 2020-09-22.

[188] Play Console Help. Prepare your app for review. https://support.google.com/googleplay/android-developer/answer/9815348?hl=en, 2020. Accessed 2020-09-25.

[189] Android Help. Download apps to your Android device. https://support.google.com/android/answer/9457058?hl=en, 2020. Accessed 2020-09-25.

[190] Jerry Hildenbrand. Sideloading and unknown sources on android: How to do it and fix errors. https://www.androidcentral.com/unknown-sources, 4 2020. Accessed 2020-09-09.

[191] F-Droid Limited and Contributors. What is F-Droid? https://f-droid.org/, 2020. Accessed 2020-09-25, mirrored for archival.

[192] Amazon. Amazon App Store for Android. https://www.amazon.com/gp/mas/get/android/, 2020. Accessed 2020-09-25.

[193] Nathan Collier. Mobile menace monday: Android trojan raises xhelper. https://blog.malwarebytes.com/android/2019/08/mobile-menace-monday-android-trojan-raises-xhelper/, 8 2019. Accessed 2020-09-09.

[194] Google. Android Security & Privacy 2017 Year in Review. TODOgithub, 3 2018. Accessed 2020-09-25, mirrored for archival.

[195] Google. Android Security & Privacy 2018 Year in Review. TODOgithub, 3 2019. Accessed 2020-09-25, mirrored for archival.

[196] Android Developers. Back up key-value pairs with Android Backup Service. https://developer.android.com/guide/topics/data/keyvaluebackup, 6 2020. Accessed 2020-09-25.

[197] Android Developers. Data backup overview. https://developer.android.com/guide/topics/data/backup, 12 2019. Accessed 2020-09-18.

[198] Android Developers. Back up user data with Auto Backup. https://developer.android.com/guide/topics/data/autobackup, 1 2020. Accessed 2020-09-25.

[199] Android Developers. Test backup and restore. https://developer.android.com/guide/topics/data/testingbackup, 12 2019. Accessed 2020-09-18.

[200] Uday Savagaonkar, Nelly Porter, Nadim Taha, Benjamin Serebrin, and Neal Mueller. Titan in depth: Security in plaintext. https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext, 8 2017. Accessed 2020-09-25.

[201] NCC Group. Android Cloud Backup/Restore. TODOgithub, 10 2018. Accessed 2020-09-25, mirrored for archival.

[202] Android Help. Back up or restore data on your Android device. https://support.google.com/android/answer/2819582, 2020. Accessed 2020-09-18.

[203] Samsung. Back up and restore data using Samsung Cloud. https://www.samsung.com/us/support/answer/ANS00060517/, 2020. Accessed 2020-09-25.

[204] LG. LG Android Backup. https://www.lg.com/us/support/help-library/lg-android-backup-CT10000025-20150104708841, 11 2018. Accessed 2020-09-25.

[205] HTC. Back up and Transfer. https://www.htc.com/us/support/backup-and-transfer/, 2020. Accessed 2020-09-25.

[206] Nischay Khanna. How to take an encrypted backup of your Android device using ADB? https://candid.technology/take-encrypted-backup-android-using-adb/, 1 2020. Accessed 2020-09-25.

[207] Ubuntu. Ubuntu manpages: adb. https://manpages.ubuntu.com/manpages/bionic/man1/adb.1.html, 2019. Accessed 2020-09-25.

[208] Wondershare. Dr. Fone Android Backup. https://drfone.wondershare.com/android-backup-and-restore.html, 2020. Accessed 2020-09-25.

[209] Anvsoft. Backup and Restore Android with SyncDroid Manager. https://www.sync-droid.com/syncdroid-getstarted.html, 2020. Accessed 2020-09-25.

[210] Syncios. iOS & Android Manager. https://www.syncios.com/features.html, 2020. Accessed 2020-09-25.

[211] Russell Brandom. This is why you shouldn't use texts for two-factor authentication. https://www.theverge.com/2017/9/18/16328172/sms-two-factor-authentication-hack-password-bitcoin, 9 2017. Accessed 2020-09-25.

[212] Raymond Leong, Dan Perez, and Tyler Dean. MESSAGETAP: Who's Reading Your Text Messages? Available at https://www.fireeye.com/blog/threat-research/2019/10/messagetap-who-is-reading-your-text-messages.html, October 2019.

[213] Patrick Siewert. Cellular Provider Record Retention Periods. https://www.forensicfocus.com/articles/cellular-provider-record-retention-periods/, 4 2017. Accessed 2020-09-25.

[214] GSMA. Universal Profile. https://www.gsma.com/futurenetworks/rcs/universal-profile/, 2020. Accessed 2020-09-25.

[215] Dieter Bohn. Google is finally taking charge of the RCS rollout. https://www.theverge.com/2019/6/17/18681573/google-rcs-chat-android-texting-carriers-imessage-encryption, 6 2019. Accessed 2020-09-25.

[216] Jibe Cloud. How RCS works. https://docs.jibemobile.com/intro/, 5 2019. Accessed 2020-09-25.

[217] Android Open Source Project. Hide RcsMessageStore APIs. https://android-review.googlesource.com/c/platform/frameworks/base/+/909259, 2 2019.

[218] Todd Haselton. Google makes texting on Android more like Apple's iMessage but with one less safeguard against spying eyes. https://www.cnbc.com/2019/12/18/google-messages-get-rcs-on-android-to-make-it-more-like-apple-imessage.html, 12 2019. Accessed 2020-09-25.

[219] Messages Help. Check your messages on your computer. https://support.google.com/messages/answer/7611075?hl=en, 2020. Accessed 2020-09-25.

[220] Iliyan Malchev. Here comes Treble: A modular base for Android. https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html, 5 2017. Accessed 2020-09-09.

[221] Android Open Source Project. Generic Kernel Image. https://source.android.com/devices/architecture/kernel/generic-kernel-image, 9 2020.

[222] Jimmy Westenberg. Bixby on the Samsung Galaxy S9: It's not all bad. https://www.androidauthority.com/samsung-galaxy-s9-bixby-850218/, 4 2018. Accessed 2020-09-09.

[223] Robert Triggs. Are Android updates getting faster? Let's look at the data. https://www.androidauthority.com/faster-android-updates-942929/, 1 2019. Accessed 2020-09-09.

[224] StatCounter Global Stats. Mobile & Tablet Android Version Market Share United States Of America. https://gs.statcounter.com/android-version-market-share/mobile-tablet/united-states-of-america, 8 2020. Accessed 2020-09-25.

[225] Andrew Laughlin. More than one billion Android devices at risk of malware threats. https://www.which.co.uk/news/2020/03/more-than-one-billion-android-devices-at-risk-of-malware-threats/, 3 2020. Accessed 2020-09-09.

[226] Google. Android Security Rewards Program Rules. https://www.google.com/about/appsecurity/android-rewards/. Accessed 2020-09-25.

[227] Whitson Gordon. Top 10 Reasons to Root Your Android Phone. https://lifehacker.com/top-10-reasons-to-root-your-android-phone-1079161983, 8 2013.

[228] Andrea Fortuna. Android Forensics: imaging android filesystem using ADB and DD. https://www.andreafortuna.org/2018/12/03/android-forensics-imaging-android-file-system-using-adb-and-dd/, 12 2019. Accessed 2020-09-25.

[229] Dallas Thomas. List of Phones with Unlockable Bootloaders . https://android.gadgethacks.com/how-to/list-phones-with-unlockable-bootloaders-0179751/, 3 2020. Accessed 2020-09-25.

[230] Android Open Source Project. Locking/Unlocking the Bootloader. https://source.android.com/devices/bootloader/locking_unlocking, 9 2020. Accessed 2020-09-25.

[231] John Wu. Magisk - The Magic Mask for Android. https://forum.xda-developers.com/apps/magisk/official-magisk-v7-universal-systemless-t3473445, 10 2016. Accessed 2020-09-25.

[232] Jorrit Jongma. firmware.mobi. https://forum.xda-developers.com/android/development/firmware-mobi-t3675896, 9 2017. Accessed 2020-09-25.

[233] XDA Developers. Best One Click Root Methods in 2020. https://www.xda-developers.com/best-one-click-root-2018/, 2020. Accessed 2020-09-25.

[234] Robert McMillan. A Jailbreak for Google's Android. https://www.pcworld.com/article/153387/article.html, 11 2008. Accessed 2020-09-25.

[235] Mark Wilson. T-Mobile G1: Full Details of the HTC Dream Android Phone. https://www.techrepublic.com/article/how-to-remove-bloatware-from-your-rooted-android-device/, 9 2008. Accessed 2020-09-25.

[236] Cameron Summerson. How to Use Android's Built-In Tethering When Your Carrier Blocks It. https://www.howtogeek.com/263785/how-to-use-androids-built-in-tethering-when-your-carrier-blocks-it/, 9 2017. Accessed 2020-09-25.

[237] Jack Wallen. How to remove bloatware from your rooted Android device. https://www.techrepublic.com/article/how-to-remove-bloatware-from-your-rooted-android-device/, 9 2014. Accessed 2020-09-25.

[238] Andrew Martonik. I haven't thought about rooting an Android phone in years — and you shouldn't either. https://www.androidcentral.com/i-havent-thought-about-rooting-my-android-phone-years, 1 2018. Accessed 2020-09-25.

[239] Samsung Research America. Samsung Knox Security Solution. TODOgithub, 5 2017. Accessed 2020-09-25, mirrored for archival.

[240] Samsung Knox Documentation. What is a Knox Warranty Bit and how is it triggered? https://docs.samsungknox.com/admin/knox-platform-for-enterprise/faqs/faq-115013562087.htm. Accessed 2020-09-25.

[241] Ryne Hager. Google's dreaded SafetyNet hardware check has been spotted in the wild. https://www.androidpolice.com/2020/06/29/googles-dreaded-safetynet-hardware-check-has-been-spotted-in-the-wild/, 6 2020. Accessed 2020-09-25.

[242] NIST National Vulnerability Database. CVE-2016-5195. https://nvd.nist.gov/vuln/detail/CVE-2016-5195, 2020. Accessed 2020-09-25.

[243] Ionut Arghire. Android Root Exploits Abuse Dirty COW Vulnerability. https://www.securityweek.com/android-root-exploits-abuse-dirty-cow-vulnerability, 10 2016. Accessed 2020-09-25.

[244] Wen Xu and Yubin Fu. Own your android! yet another universal root. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association.

[245] Catalin Cimpanu. Samsung patches 0-click vulnerability impacting all smartphones sold since 2014. https://www.zdnet.com/article/samsung-patches-0-click-vulnerability-impacting-all-smartphones-sold-since-2014/, 5 2020. Accessed 2020-09-26.

[246] Guang Gong. TiYunZong: An Exploit Chain to Remotely Root Modern Android Devices. TODOgithub, 8 2020. Accessed 2020-09-26, mirrored for archival.

[247] Grant Hernandez. Tailoring CVE-2019-2215 to Achieve Root. https://hernan.de/blog/2019/10/15/tailoring-cve-2019-2215-to-achieve-root/, 10 2019. Accessed 2020-09-26.

[248] NIST National Vulnerability Database. CVE-2017-13174. https://nvd.nist.gov/vuln/detail/CVE-2017-13174, 2019. Accessed 2020-09-25.

[249] NIST National Vulnerability Database. CVE-2017-5947. https://nvd.nist.gov/vuln/detail/CVE-2017-5947, 2019. Accessed 2020-09-25.

[250] Roee Hay and Noam Hadad. Exploiting Qualcomm EDL Programmers (1): Gaining Access & PBL Internals. https://alephsecurity.com/2018/01/22/qualcomm-edl-1/, 1 2018. Accessed 2020-09-25.

[251] Roee Hay and Noam Hadad. Exploiting Qualcomm EDL Programmers (2): Storage-based Attacks & Rooting. https://alephsecurity.com/2018/01/22/qualcomm-edl-2/, 1 2018. Accessed 2020-09-25.

[252] bovirus. MTK Android (SP Flash Tool) Tutorial. https://forum.xda-developers.com/showthread.php?t=1982587, 11 2012.

[253] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, et al. Attention spanned: Comprehensive vulnerability analysis of {AT} commands within the android ecosystem. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 273–290, 2018.

[254] awakened1712. How a double-free bug in WhatsApp turns to RCE . https://awakened1712.github.io/hacking/hacking-whatsapp-gif-rce/, 10 2019.

[255] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*, pages 18–20, 2020.

[256] Brian Carrier. Open Source Digital Forensics. `https://www.sleuthkit.org/`, 2020. Accessed 2020-09-25.

[257] Shiv Mel. Rooting for Fun and Profit: A study of PHA exploitation in Android. `TODOgithub`, 2016. Accessed 2020-09-25, mirrored for archival.

[258] Magnet Forensics. Advanced Mobile Acquisition for Android. `https://www.magnetforensics.com/resources/advancedmobile/`. Accessed 2020-09-25.

[259] Google. Global requests for user information. `https://transparencyreport.google.com/user-data/overview`. Accessed 2020-09-25.

[260] Jennifer Valentino-DeVries. Google's Sensorvault Is a Boon for Law Enforcement. This Is How It Works. `https://www.nytimes.com/2019/04/13/technology/google-sensorvault-location-tracking.html`, 4 2019. Accessed 2020-09-25.

[261] Gabriel J.X. Dance and Jennifer Valentino-DeVries. Have a Search Warrant for Data? Google Wants You to Pay. `https://www.nytimes.com/2020/01/24/technology/google-search-warrants-legal-fees.html`, 1 2020. Accessed 2020-09-25.

[262] Steven Levy. How Google is Remaking Itself as a "Machine Learning First" Company. `https://www.wired.com/2016/06/how-google-is-remaking-itself-as-a-machine-learning-first-company/`, 6 2016. Accessed 2020-09-28.

[263] Shanhong Liu. Android operating system share worldwide by OS version from 2013 to 2020. `https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/`, 4 2020. Accessed 2020-11-07.

[264] Google Project Zero. Project zero issue tracker. `https://bugs.chromium.org/p/project-zero/issues/list`. Accessed 2020-11-07.

[265] Dawinder S Sidhu. The chilling effect of government surveillance programs on the use of the internet by muslim-americans. *U. Md. LJ Race, Religion, Gender & Class*, 7:375, 2007.

[266] Jonathon W Penney. Chilling effects: Online surveillance and wikipedia use. *Berkeley Tech. LJ*, 31:117, 2016.

[267] Margot E Kaminski and Shane Witnov. The conforming effect: First amendment implications of surveillance, beyond chilling speech. *U. Rich. L. Rev.*, 49:465, 2014.

[268] Alex Marthews and Catherine E Tucker. Government surveillance and internet search behavior. *Available at SSRN 2412564*, 2017.

[269] Dirk Van Bruggen. *Studying the impact of security awareness efforts on user behavior*. PhD thesis, University of Notre Dame, 2014.

[270] M. Schulz and J.A. Hennis-Plasschaert. REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL. `https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&from=EN#d1e6620-1-1`, 4 2016. Presidents of European Parliment and EU Council listed. Accessed 2020-10-25.

[271] Ivan Cherapau, Ildar Muslukhov, Nalin Asanka, and Konstantin Beznosov. On the Impact of TouchID on iPhone Passcodes. In *Eleventh Symposium On Usable Privacy and Security ({SOUPS} 2015)*, pages 257–276, 2015.

[272] David Brash. Armv8-A architecture: 2016 additions. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-2016-additions, 10 2016. Accessed 2020-07-19.

[273] Android Developers. getExternalStorageDirectory. https://developer.android.com/reference/android/os/Environment.html#getExternalStorageDirectory(), 6 2020. Accessed 2020-09-09.

[274] Paul Lawrence. Seccomp filter in Android O. https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html, 7 2017. Accessed 2020-09-09.

[275] Android Developers. Behavior changes: all apps. https://developer.android.com/about/versions/pie/android-9.0-changes-all, 12 2019. Documentation for Android 9, accessed 2020-09-09.

[276] Corbin Davenport. Google Play Store will make app bundles a requirement in 2021. https://www.androidpolice.com/2020/06/12/google-play-store-will-make-app-bundles-a-requirement-in-2021/, 6 2020. Accessed 2020-11-09.

[277] Mark Murphy. Uncomfortable Questions About App Signing. https://commonsware.com/blog/2020/09/23/uncomfortable-questions-app-signing.html, 9 2020. Accessed 2020-09-25.

[278] Google. Android Compatibility Definition. TODOgithub, 9 2020. Accessed 2020-09-09, mirrored for archival.

[279] Liang Kai. Guard your data with the Qualcomm Snapdragon Mobile Platform. TODOgithub, 4 2019. Accessed 2020-09-09, mirrored for archival.

[280] Meltem Sönmez Turan, Elaine B. Barker, William E. Burr, and Lidong Chen. Sp 800-132. recommendation for password-based key derivation: Part 1: Storage applications. Technical report, U.S. Government, Gaithersburg, MD, USA, 2010.

Appendix A

History of iOS Security Features

A.1 iOS Security Features Over Time

A.1.1 Encryption and Data Protection over time

In 2009, iOS 3 was released on the iPhone 3GS with the first iteration of Data Protection, which entailed encrypting the flash storage of the device when powered off, primarily to facilitate rapid erasure of the device. In this initial release, the user passcode was *not* mixed into the encryption key and thus was not necessary for decryption. In 2014, iOS 8 (released along with the iPhone 6 and 6+) increased the coverage of data encrypted using a key derived from the user's passcode, protecting significantly more data with keys unavailable without the user's consent. This decision drew significant criticism from law enforcement, most notably in an early speech by former FBI Director James Comey [8]. Since its inception, Data Protection classes have been introduced and applied to various data as summarized in Table 3.10. With Data Protection, when files are deleted, their keys are evicted and as such this data is unrecoverable, however, in some cases user-initiated delete options simply move data to a "deleted" section of a SQL database on-device which may remain recoverable [113].

A.1.2 Progression of passcodes to biometrics

In early versions of iOS, passcodes defaulted to four digits and were enforced by software running on the application processor. This provided a superficial layer of security, which could potentially be bypassed through a software exploit against the operating system. With the iPhone 5S and iOS 9 in 2015, Apple increased the default for new passcodes to 6 digits while also introducing TouchID, a capacitive fingerprint sensor. In 2017 with the iPhone X, some models of iPhone replaced TouchID with FaceID, a facial recognition biometric, similarly authenticated through the SEP. Apple argues that these ease-of-use features improve security by enabling users to employ higher-quality passcodes (as the biometric sensor reduces the frequency at which they must be entered) [73]; the efficacy of this decision has questioned in the scientific literature [271].

A.1.3 Introduction of the Secure Enclave

Early iOS devices performed most security functions solely using the application processor, which is the same processor that runs the operating system and all application software. As of the iPhone 5S in 2013, Apple began shipping an additional component in iOS devices: the Secure Enclave Processor, or SEP [27, 28]. The SEP architecture and hardware has undergone a few improvements over time as documented in Tables 3.10 and 3.11.

A.1.4 Hardware changes for security

Just as the software, iOS, has evolved year to year, so too has iPhone hardware. Early generations integrated commodity components. However, recent iPhone and iOS devices employ custom hardware and processor designs [93]. Table 3.11 summarizes changes to iPhone hardware over time which are directly security-relevant, and the remainder of this section covers these changes in detail.

From iPhone 2G to 3GS (2007–2009), Apple sourced hardware from Samsung, LG, and other manufacturers to make their phone. Few if any security-specific hardware features were present in these early phones, and this remained true even as Apple transitioned to in-house designed System-on-Chips (SoCs) with the A4 and continued with the A5 and A6 [86, 87, 88]. These early iterations encrypted the boot NOR memory with a hardware-fused AES key (UID key), driven by a cryptographic accelerator known as the Crypto Engine.[1] The A7 in iPhone 5S introduced the Secure Enclave Processor (SEP) and a new fingerprint sensor in the home button to enable TouchID. The SEP at this iteration included its own UID key used for encryption. Enclave memory is encrypted with a key derived from both the UID key and an ephemeral key stored in the enclave, and enclave memory is authenticated on A8 and later. The A9 SoC in the iPhone 6S/6S+ connected the path between flash storage and memory directly via the Crypto Engine, in order to improve file encryption security and performance, and allows the SEP to generate its UID key rather than be fused with it. SEP key generation also has the advantage that not even the manufacturers know the SEP UID key. The A10 SoC uses the SEP to protect filesystem class keys in Recovery Mode. The A11 SoC present in the iPhone X added the Neural Engine, a CPU component which accelerates neural network computation, and enables FaceID by allowing facial recognition authentication in the SEP to directly access the Neural Engine [73]. A11 also introduces the "integrity tree" for stronger authentication of enclave memory. The A12 in XS, XS Max, and XR adopted a version of Pointer Authentication Codes (PAC) from ARM v8.3 [272] with the relevant hardware support as an additional defense against memory corruption exploits, and added a "secure storage integrated circuit" for the SEP to use for replay counters, tamper detection, and to aid cryptography and random number generation. Apple notes that the enclave communicated with the secure storage IC via a "secure protocol." The A12 also extended Recovery Mode protections to DFU mode.

[1] The Crypto Engine is documented as early as the A5 SoC, but may have been included earlier

A.1.5 Moving secrets to the cloud

iCloud Keychain was announced in 2016 [72], and then in 2017 with iOS 11 on iPhones 8, 8 Plus, and X CloudKit containers extended an API to third-party developers for them to store arbitrary data encrypted with keys in the user iCloud Keychain (thus not decrypt-able by Apple) called CloudKit [52]. Apple additionally introduced the use of CloudKit for iMessage in this time range [28, 31].[2] The historical development of iCloud security is not extensively documented by Apple, but §3.1 provides an overview of the current documented architecture in depth.

[2]With the aforementioned caveat regarding iCloud Backup

Appendix B

History of Android Security Features

B.1 Android Security Features Over Time

B.1.1 Application sandboxing

The original sandbox on Android was filesystem-based, with applications sandboxed from each other via user IDs and permissions. The filesystem on Android has historically been divided into "internal" (device-specific) and "external" (non-device-specific). The external storage, also known by its directory name of `/sdcard`, holds arbitrary files, such as music, photos, and documents, that the user can use across several apps or even off the device itself. Internal storage is controlled closely by the Android system – each app gets access to its own part of internal storage – but external storage has fewer controls, because of its intention as a shared resource. Traditionally, external storage was a removable flash memory device (such as an SD Card), but the term has evolved to encompass all shared storage on Android, even for those devices without a removable memory slot [273].

Android 7.0 began locking down access to `/sdcard`, changing directory access control (DAC) permissions from 751 (Unix `rwxr-x--x`) to 700 (Unix `rwx------`) and removing the ability for apps to request explicit URIs to other app's files. Android 10 took this a step further, deprecating raw access to /sdcard and only allowing the system explicit access to *only* files that it has created. All other views of the file system are mediated by the operating system, and require user intervention [133]. Android 11 enforces scoped storage, eliminating `/sdcard` access entirely [145].

An augmentation to the existing filesystem-based sandbox came in the form of SELinux, which was first added to the Linux kernel with Android 4.3 in permissive (logging) mode, and Android 4.4 in enforcing (blocking) mode. Originally, only a subset of domains (SELinux protection types) were utilized for critical system functions; Android 5.0 pushed SELinux enforcement to the rest of the system. Android 6.0 and 7.0 continued to strengthen the SELinux sandbox, tightening the scope of domains and breaking up services to isolate permissions to specific subsystems. Android 8.0 adds updates for Android's Project Treble hardware abstraction initiative, and 9.0 requires that all apps run in individual SELinux sandboxes (and thus preventing apps from making data world-accessible) [146, 133].

Dovetailing with SELinux is `seccomp-bpf`, another sandboxing feature Android inherits from the upstream Linux kernel. This facility, added to Android in version 8.0, prevents applications from making syscalls explicitly whitelisted. This whitelist was built from syscalls available to applications (via the Bionic C interface), syscalls required for boot, and syscalls required for popular apps [274]. Android 9.0 further improves on the filter, paring down the whitelist even further [275].

Application permissions to media, dialer, messages, *etc.*, are important to the security of the Android system, but not relevant to the discussion of this work.

B.1.2 Data storage & encryption

On release, Android did not employ any encryption mechanisms to protect user data. Android 4.4 introduced full-disk encryption, but it would not be until Android 5.0 that it would become the default. Full-disk encryption uses the PIN/password (4.4) or pattern (5.0) to encrypt the encryption key, and stores it in a secure hardware location if available (5.0). Note that this encryption provides protection only for lost or stolen devices, as once the device is powered on, the device remains decrypted [141]. Android 7.0 introduced file-based encryption, discussed above, and Android 10 made it mandatory for all devices going forward [138]. File-based encryption allows devices to Direct Boot, allowing critical system functions to run without the phone unlocking [161]. The prior full-disk encryption used the Linux kernel's `dm-crypt` block-level encryption, meaning that the user's passcode was required before any data could be unlocked. Encryption of file metadata was integrated into Android 9,0. The combination of file-based encryption and metadata encryption is thus equivalent to full-disk encryption [143].

B.1.3 Integrity checking

Starting in version 4.4, Android validated boot code via the `dm-verity` Linux kernel module, which uses a secure hardware-backed hash tree to validate data on a partition. Any device corruption would be detected and presented to the user as a warning, but the device would still be allowed to boot. It would not be until Android 7.0 that Verified Boot would be strictly enforced, preventing boot if the `dm-verity` check fails. A new version of Verified Boot, known as Android Verified Boot, was introduced in version 8.0 alongside the changes for Project Treble. AVB also added rollback protection via features in the TEE [148]. Note that this integrity check is disabled when a user roots their device via an unlocked bootloader.

In addition to the checks performed at boot time, the Android system validates the APK (Android Package) of an app before installation; if the signature on the APK does not verify with its public key, Android refuses to install it. Android 2.1 introduced the first version APK verification, using the basic Java JAR verification mechanism. Android 7.0 introduced an APK-specific signing scheme, and Android 9.0 improved upon it by allowing for key rotation on APK updates [134]. Note that this is separate from the installation of apps from trusted sources (such as Google Play), which is a separate mechanism [186].

Google has since introduced App Bundles, which consist of pre-compilation code and resources. Google has mandated that all new Android apps on the Play Store use App Bundles some time in 2021 [276]. Before an app is published on Google Play, Google builds and signs the final APK bundle. Rather than allowing developers to directly sign APKs, App Bundles shift control of final signing keys to Google. This move has raised questions from app developers who protest that Google is now empowered to modify apps before distributing them, potentially to inject Google advertising, or at the request of overreaching governments [277].

B.1.4 Trusted hardware

The history of trusted hardware on Android is inherently more complex, as the Open Handset Alliance does not mandate the use of any specific hardware for Android [278], and there is no requirement in Android that the TEE be an ARM TrustZone OS, just that it supports the relevant hardware features [278]. However, Qualcomm's QSEE environment meets Android requirements [279], and Google provides a reference TEE, Trusty, for general use [147].

One of the early uses of TEEs in Android was for key material. Starting in version 4.1, Android provided a Keymaster Hardware Abstraction Layer (HAL) to store keys and perform signing and verifying operations in a trusted application (Keymaster TA). Successive Android versions increased the capabilities of the Keymaster TA. Android 6.0 added symmetric features (AES and HMAC) and key usage control. Keymaster 2, released with Android 7.0, added key attestation, allowing remote clients to verify that a key was stored in a Keymaster TEE, as well as version bindings, preventing version rollbacks. Keymaster 3 (Android 8.0) extended the attestation features provided by the underlying TEE to also provide ID attestation using the phone's IMEI and hardware serial number. Keymaster is not the only Android system that uses the TEE: the Gatekeeper and Fingerprint user authenticator, mentioned previously, are also implemented in trusted applications. The authentication components use TEE features to store passwords and authentication data [142].

Recent years have seen Android manufacturers adding additional hardware security modules to their devices. This module, known as a *secure element*, is an analogue to the Secure Enclave in iOS: a dedicated cryptographic module, separate from the primary processor, for computation on secret data. The Android OS view of the use of secure elements is through StrongBox Keymaster interface, added in Android 9. This extends the Keymaster TA functions to support cryptographic operations off the main system processor [144].

Appendix C

History of Forensic Tools

In the following Tables C.1, C.2, C.3, and C.4, we enumerate the forensic software tools evaluated by DHS as of this writing. These tables are divided by year for readability.

Date	**iPhone (iOS)**[a]	**Android**[b]	**Forensic Tool**	**Version**
Sep '19	7 Plus (10.2)	7.1.1	UFED InField Kiosk	7.5.0.875
Jun '19	X and 8 Plus (11)[c]	-	Paraben's Electronic Evidence Examiner – Device Seizure	2.2.11812.15844
Jun '19	X and 8 Plus (11)[d]	N/A[e]	GrayKey OS/App Bundle	1.4.2/1.11.2.5
Apr '19	X and 8 Plus (11)[f]	8.1.0	UFED 4PC/Physical Analyzer	7.8.0.942

Table C.1: History of Forensic Tools (2019)

Source: DHS [22]

[a]Latest model/version included in test.
[b]Latest version included in test.
[c]The iPhone X ran iOS 11.3.1, the iPhone 8 ran 11.4.1.
[d]X (11.3.1), 8 (11.4.1).
[e]iOS-only tool.
[f]X (11.3.1), 8 (11.4.1).

Date	iPhone (iOS)[a]	Android[b]	Forensic Tool	Version
Nov '18	7 Plus (10.2)	7.1.1	Blacklight 2018	1.1
Nov '18	7 Plus (10.2)	7.1.1	Mobilyze	2018.1
Nov '18	7 Plus (10.2)	7.1.1	XRY	7.8.0
Nov '18	7 Plus (10.2)	-	XRY Kiosk	7.8.0
Oct '18	7 Plus (10.2)	7.0	Magnet AXIOM	1.2.1.6994
Jul '18	7 Plus (10.2)	6.0.1	Final Mobile Forensics	2018.02.07
Jun '18	7 Plus (10.2)	7.1.1	Electronic Evidence Examiner - Device Seizure	1.7
May '18	7 (10.2)	7.1.1	Katana Forensics Triage	1.1802.220
Apr '18	7 Plus (10.2)	7.1.1	MD-NEXT/MD-RED	1.75/2.3
Apr '18	7 Plus (10.2)	7.1.1	Oxygen Forensics	10.0.0.81
Jan '18	7 Plus (10.2)	7.1.1	MOBILedit Forensics	9.1.0.22420
Jan '18	7 Plus (10.2)	7.1.1	UFED Touch/Physical Analyzer	6.2.1.17

Table C.2: History of Forensic Tools (2018)

Source: DHS [22]

[a]Latest model/version included in test.
[b]Latest version included in test.

Date	iPhone (iOS)[a]	Android[b]	Forensic Tool	Version
Dec '17	7 Plus (10.2)	7.1.1	Blacklight	2016.3.1
Dec '17	7 Plus (10.2)	7.1.1	Mobilyze	2017.1
Nov '17	7 (10.2)	7.1.1	Secure View	4.3.1
Nov '17	7 Plus (10.2)	7.1.1	MOBILedit Forensics Express	4.2.1.11207
Sep '17	7 Plus (10.2)	7.1.1	UFED 4PC/Physical Analyzer	6.2.1
Aug '17	7 Plus (10.2)	7.1.1	XRY Kiosk	7.3.0
Aug '17	7 Plus (10.2)	7.1.1	XRY	7.3.1
Jul '17	7 (10.2)	7.1.1	Lantern	4.6.8
Jun '17	7 (10.2)	7.1.1	Final Mobile Forensics	2017.02.06
Apr '17	6S (9.2.1)	5.1.1	Electronic Evidence Examiner Device Seizure	1.0.9466.18457
Mar '17	6S (9.2.1)	5.1.1	Mobile Phone Examiner Plus	5.6.0
Mar '17	6S (9.2.1)	5.1.1	MOBILedit Forensic Express	3.5.2.7047
Jan '17	6S (9.2.1)	5.1.1	XRY Kiosk	7.0.0.36568

Table C.3: History of Forensic Tools (2017)

Source: DHS [22]

[a]Latest model/version included in test.
[b]Latest version included in test.

Date	iPhone (iOS)[a]	Android[b]	Forensic Tool	Version
Dec '16	6S (9.2.1)	-	MOBILedit Forensic	8.6.0.20354
Nov '16	6S (9.2.1)	5.1.1	MOBILedit Forensic	8.6.0.20354
Nov '16	6S (9.2.1)	-	XRY	7.0.1.37853
Aug '16	6S (9.2.1)	5.1.1	Oxygen Forensics	8.3.1.105
Jul '16	6S (9.2.1)	-	Secure View	4.1.9
Jul '16	5S (7.1)	-	UFED Touch	4.4.0.1
May '16	6S (9.2.1)	-	Device Seizure	7.4
May '16	6S (9.2.1)	5.1.1	BlackLight	2016.1
Jan '16	5S (7.1)	-	UFED 4PC/Physical Analyzer	4.2.6.4-5
Dec '15	5S (7.1)	-	MOBILedit Forensic	7.8.3.6085
Dec '15	5S (7.1)	-	Phone Forensics Express	2.1.2.2761
Jun '15	5S (7.1)	-	Device Seizure	6.8
Apr '15	5S (7.1)	-	EnCase Smartphone Examiner	7.10.00.103
Jun '15	5S (7.1)	-	Lantern	4.5.6
Mar '15	5S (7.1)	-	Oxygen Forensic Analyst	7.0.0.408
Feb '15	5S (7.1)	-	Secure View	3.16.4
Dec '14	5S (7.1)	N/A	iOS Crime Lab	1.0.1
Dec '14	5S (7.1)	-	Mobile Phone Examiner Plus	5.3.73
Oct '14	5S (7.1)	-	UFED Physical Analyzer	3.9.6.7
Sep '14	5S (7.1)	-	XRY/XACT	6.10.1
Apr '13	4 (5.0.1)	-	EnCase Smartphone Examiner	7.0.3
Feb '13	4 (5.0.1)	2.2.1	Device Seizure	5.0
Feb '13	4 (5.0.1)	-	Micro Systemation XRY	6.3.1
Feb '13	4 (5.0.1)	N/A	Lantern	2.3
Feb '13	4 (4.3.3)	2.1.1	Secure View 3	3.8.0
Sep '12	4 (4.3.3)	2.1.1	Mobile Phone Examiner Plus	4.6.0.2
Sep '12	4 (4.3.3)	2.1.1	CelleBrite UFED	1.1.8.6
Jan '11	3GS (3.0)	N/A	Mobilyze	1.1
Dec '10	3GS (3.0)	N/A	Zdziarski's Method	N/A
Dec '10	3GS (3.0)	N/A	iXAM	1.5.6
Nov '10	3GS (3.0)	1.5	Secure View	2.1.0
Nov '10	3GS (3.0)	1.5	XRY	5.0.2

Table C.4: History of Forensic Tools (2010–2016)

Source: DHS [22]

[a]Latest model/version included in test.
[b]Latest version included in test.

Glossary

2FA Two-Factor Authentication (2FA) is a mechanism used to augment traditional password-based authentication to remote services, by adding an additional "factor" of authentication related to *e.g.,* biometrics or possession of a device. 20, 35

AES Advanced Encryption Standard (AES) is a cipher accepted by the National Institute of Standards and Technology, widely deployed and used to protect the confidentiality of digital data at rest and in transit. 15, 18, 24, 29, 50

AFU After First Unlock (AFU) is an Apple-specific term given to the state of a device that has been powered on, and into which the user has entered their passcode at least once. Devices remain in this state until they are manually rebooted, or some defined time period has elapsed. 4, 16, 17, 24, 29, 31, 35, 36, 39–41, 43, 44, 51

AOSP Android Open Source Project, an open source software project defining the non-proprietary elements of the Android operating system. AOSP is produced by the Open Handset Alliance, with sponsorship from Google.. 7, 48, 49, 52, 53, 55, 56, 58–60, 62–64

API Application Programming Interfaces (API) represent a set of subroutines that provide system services to application developers. 14, 15, 17, 19, 37, 48, 53, 54, 68, 97

ARM Arm (previously ARM, for Advanced RISC Machine, and further prior, Acorn RISC Machine) is a family of reduced instruction set architectures primarily designed for mobile and embedded systems and SoCs. 9, 25, 52, 96, 100

AVB Android Verifiable Boot is a service for verifying the integrity of the Android boot chain using a TEE.. 53, 63, 99

CBC Cipher Block Chaining (CBC) is a cryptographic algorithm which enables encryption of a data stream using AES. 24, 50

CP CP or "Complete Protection" is the strongest class of Data Protection on iOS. Data which is CP is encrypted in memory, and encryption keys are evicted from memory shortly after the device is locked.. 16, 24, 29, 31, 43

CPU Central Processing Unit, the main computing hardware within a computer. 25

Curve25519 Curve25519 is a mathematical construction used in ECDH and other cryptographic algorithms. 15, 17

DFU Device Firmware Upgrade (DFU) mode is a special device state in which firmware such as the bootloader may be upgraded or modified. 35

DHS The U.S. Department of Homeland Security (DHS) is one agency responsible for evaluating forensic tooling for the U.S. Federal government. 2, 4, 6, 11, 26, 36, 42, 43, 72, 73, 101–104

ECDH Elliptic Curve Diffie-Hellman (ECDH) is a cryptographic algorithm which enables secure two-party key agreement. 15

FaceID A facial-recognition biometric authentication system developed by Apple for iOS devices (mostly replacing TouchID).. 13, 21, 22, 24, 25, 44

FaceTime FaceTime is Apple's end-to-end encrypted video and audio chat service. FaceTime connects registered Apple users to enable real-time audio and video communication.. 22, 23, 27

FBI The U.S. Federal Bureau of Investigation (FBI) is a governmental agency which serves as the principle law enforcement office, as well as a domestic security and intelligence organization. 1, 43, 73

GCM Galois-Counter Mode (GCM) is a cryptographic algorithm which enables authenticated encryption of a data stream using AES. 24

HSM A Hardware Security Module (HSM) is a specialized computing device designed to store and operate on cryptographic secrets. These devices often include software and physical protection mechanisms intended to provide strong security for secret keys, while making these keys usable to authorized systems. Some HSMs can be provisioned with custom software. 5, 6, 20, 45, 57

iMessage iMessage is Apple's end-to-end encrypted messaging service. iMessage delivers text and media via signed and encrypted messages among registered Apple users.. 16–18, 22, 24, 27

jailbreak An exploit or software package which bypasses iOS security to enable unsigned code execution and custom kernel modifications.. 7, 14, 26–31, 33, 35, 44, 45

JTAG Joint Test Action Group. An IEEE standard for, and implementation of, hardware debugging via on-chip pin interfaces. In practice, also a potential point of failure used to bypass low-level (bootloader, OS, etc) security systems.. 32, 46

kernel The core software component of a standard computer operating system, responsible for maintaining privilege separation between applications and processes on a device.. 6, 21, 26, 28, 30, 31, 35, 37, 41, 50, 52, 55, 65, 66, 74

Lightning Lightning is a hardware implementation and proprietary standard developed by Apple for iOS devices; Lightning is used for charging and data transfer. 22, 24, 28, 37, 45, 46

MMS Multimedia Messaging Service (MMS) is a media messaging protocol and service which operates over cellular phone networks to transmit photos, videos, or audio. 11, 18, 22, 36

NIST The National Institute of Standards and Technology (NIST) is a U.S. government organization responsible for standards across industries including software, hardware, and technology. 5, 10, 11, 15, 36

PBKDF2 Password-Based Key Derivation Function 2 (PBKDF2) [280] is a cryptographic algorithm which iteratively applies pseudo-random functions to an input. This can enable storage and comparison of values without storing potentially sensitive inputs in long-term storage, among other functionalities. 17, 45

RCS Rich Communication Services (RCS) is a messaging protocol that is designed as a replacement for traditional SMS/MMS communications.. 5, 7, 59, 60, 73

RNG A Random Number Generator (RNG) is a cryptographic algorithm for generating random or pseudo-random data for use in further cryptographic algorithms. 25

sandbox A software-enforced set of access controls which prevent an application or service from accessing data, system resources, or hardware.. 14

SEP The Apple Secure Enclave Processor (SEP) is a specialized co-processor included in Apple iPhones since the iPhone 5S. This processor is designed to store and employ cryptographic secrets, and to handle authentication procedures using FaceID and TouchID. The SEP is designed to withstand attacks that result in a total compromise of the device operating system. 5, 21, 24–26, 29–31, 35, 38, 40, 41, 47

Siri Siri is the name for Apple's voice recognition system.. 19

SMS Short Message Service (SMS) is a text-based message service which operates over cellular phone networks. 11, 18, 22, 36, 59, 60

SoC System-on-a-Chip (SoC) is a hardware configuration wherein a central processor and processor peripherals are included within a single silicon package, often appropriate for embedded or mobile devices. 21, 25, 52

TA A Trusted Application is a software component that runs within a Trusted Execution Environment in order to realize a specific service.. 52

TEE A Trusted Execution Environment (TEE) is a software component that runs within a dedicated isolated execution mode on a processor. It enables the storage and use of cryptographic secrets, which remain isolated from the operating system and application software.. 51

TLS Transport Layer Security (TLS) is a cryptographic protocol for securing connections between devices on a network.. 54

TouchID A fingerprint biometric authentication system developed by Apple for iOS devices (and later for macOS computers).. 13, 21, 22, 24, 25

UID key The UID key is a random AES key that is stored within the device silicon at manufacture time via fuses; this key is used as an ingredient in deriving cryptographic keys for Apple's file encryption. 15–18, 25, 29

USB Universal Serial Bus (USB) is a hardware implementation and standard designed to enable universal device-to-device or client-and-host communication via a cable. 22, 24, 28, 35, 39, 44–46, 58, 65, 68

VPN A Virtual Private Network (VPN) serves as a proxy to connect to the internet via the VPN servers. Connections to VPN servers are often encrypted and authenticated. VPNs can protect users from surveillance by their Internet Service Provider, but this protection is only as good as the VPN service. 16

XTS XOR-Encrypt-XOR-based tweaked-codebook mode with ciphertext stealing (XTS) is a cryptographic algorithm which enables encryption of a filesystem with AES, providing stronger authentication properties than CBC and some other modes, but lacking the security associated with true authenticated encryption. 24, 50