RICE UNIVERSITY

Identifying and Mitigating Misuse of Secrets in Android with Dynamic Analysis Techniques

by

Jaeho Lee

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Dan S. Wallach, Chair
Professor of Computer Science

Ang Chen
Assistant Professor of Computer Science

Nathan Dautenhahn
Assistant Professor of Computer Science

Ray Simar
Professor in the Practice of Electrical and Computer Engineering

Houston, Texas

November, 2019
ABSTRACT

Identifying and Mitigating Misuse of Secrets in Android with Dynamic Analysis Techniques

by

Jaeho Lee

Mobile phones have been completely changing the way people think and behave, making our lives convenient. At the same time, this accelerated growth has brought with it unprecedented new threats related to user privacy, attracting hackers with various sensitive data on the phone. However, it is challenging to secure user data in Android devices because the Android framework is complex with multiple software layers, hindering security experts from implementing necessary security features. Also, a myriad of apps in Android are handling various user data, and each app developer has the principle responsibility to protect them. However, developers have varying levels of secure coding practice, and the resulting apps may inadvertently misuse of user sensitive data.

This dissertation presents studies with various Android apps and the Android framework to understand the misuse of secrets in the mobile environment. To assist my work, I have used various analysis techniques and developed a dynamic analysis framework to perform systematic analyses of Android apps. This dissertation describes approaches and my findings on how sensitive data is misused, and tools and solutions I have developed to mitigate the found security problems.

My research has had a practical impact on the industry and helped to mitigate the misuse of secrets in the mobile ecosystem. Specifically, I have discovered various patterns of
mistakes in the Android framework and popular apps that can be exploited by adversaries. For instance, I have developed FlowPass, an efficient and informative dynamic taint tracking system, which found 13 previously unknown security bugs in popular apps that have each been installed more than one million times. I have reported these misuses to the app vendors, and most have fixed the bugs shortly afterward.
Acknowledgments

This dissertation wouldn’t have existed had I not been able to surround with brilliant role models and devoted family members.

Professor Dan Wallach is the perfect advisor for me. I’m thankful that Dan always leads me in the right direction with his vast knowledge. Besides research, Dan has shown me the type of advisor and mentor that I want to be, valuing integrity and the freedom of research than productivity. I’m grateful to meet him and will reserve my deepest gratitude for his guidance and support. I also would like to thank Professor Tracy Volz for teaching me how to present, welcoming our family which helped to settle abroad.

It was my luck that Professor Ang Chen joined Rice University after my second year, and I was able to work together. I would like to thank Ang for his sincere advice, especially when I had the hardest time physically and emotionally. It was all fantastic moments for me to discuss and write papers together with Ang. He shows me what perfection is. I would like to thank Professor Nathan Dautenhahn for advising me with his in-depth knowledge, insight on the system. Without his feedback, I wouldn’t have spotted overlooked points in my research. I would like to thank Professor Ray Simar for questions, feedback throughout my thesis defense.

There is one particular individual that I owe a very big thank-you to; my soul mate and wife, Hyeju. Whenever I submit a paper after hard work, she becomes exhausted more than me, proving how hard her support and waiting are. She is the one who has empowered me to pursue my goals, and encouraged me to reach my highest potential. Her words of wisdom, endless love, and support made my journey of Ph.D. finished successfully. I am grateful to my kids, Soohyun and Soomin, for being happy and fueling me with love and joy, and their patience during my thesis work.
I also want to thank my father and mother for being healthy and always showing faith in me. My mother’s unconditional love helps me to overcome hardships with a positive attitude. My father’s dedication and hard work to raise our family get me to stand up again whenever I wanted to give up and be lazy.

Last but not least, I would like to thank my father-in-law for overcoming cancer and advising me of his wisdom. His motto of ‘the right behavior, the right way’ always lead me to choose the right direction. He is the role model of my life as the strongest father of daughters.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations xi
List of Tables xiii

1 Introduction 1
1.1 My Thesis ......................................................... 1
1.2 Contribution and Outline ................................. 3
   1.2.1 Removing secrets from Android’s TLS ............... 3
   1.2.2 Total Recall: Persistence of Password in Android ....... 4
   1.2.3 FlowPass: Understanding password misuse in Android .... 4

2 Removing Secrets from Android’s TLS .................. 6
2.1 Introduction ....................................................... 6
2.2 Background ....................................................... 9
   2.2.1 TLS .............................................................. 9
   2.2.2 Android application lifecycle ....................... 12
   2.2.3 TLS Implementation on Android .................... 15
2.3 Black-box security analysis ............................... 15
   2.3.1 Methodology ................................................. 15
   2.3.2 Test Framework ........................................... 17
   2.3.3 Supporting memory dump on Android devices .......... 18
   2.3.4 HTTPS client on Android devices .................. 18
   2.3.5 Server ....................................................... 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.6</td>
<td>Experiment</td>
<td>19</td>
</tr>
<tr>
<td>2.3.7</td>
<td>Results</td>
<td>21</td>
</tr>
<tr>
<td>2.3.8</td>
<td>Observation and Raised Questions</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>In-depth Analysis of Android Framework</td>
<td>25</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Overview</td>
<td>25</td>
</tr>
<tr>
<td>2.4.2</td>
<td>BoringSSL</td>
<td>27</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Conscrypt</td>
<td>29</td>
</tr>
<tr>
<td>2.4.4</td>
<td>OkHttp</td>
<td>34</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Summary of the problem</td>
<td>35</td>
</tr>
<tr>
<td>2.5</td>
<td>Evaluation of Attack Feasibility</td>
<td>36</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Threat model</td>
<td>37</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Extracting a master secret</td>
<td>38</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Measuring secret retention in active use</td>
<td>39</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Decrypting TLS communication</td>
<td>41</td>
</tr>
<tr>
<td>2.6</td>
<td>Discussion</td>
<td>42</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Solutions</td>
<td>42</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Observations and Future Work</td>
<td>46</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Android 8</td>
<td>48</td>
</tr>
<tr>
<td>2.7</td>
<td>Related Work</td>
<td>49</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Memory Forensics</td>
<td>49</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Android and TLS Security</td>
<td>50</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Mitigation for memory disclosure attack</td>
<td>50</td>
</tr>
<tr>
<td>2.8</td>
<td>Conclusion</td>
<td>51</td>
</tr>
<tr>
<td>3</td>
<td>Total Recall: Persistence of Password on Android</td>
<td>53</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>53</td>
</tr>
<tr>
<td>3.2</td>
<td>Background and Motivation</td>
<td>57</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Authentication in Android</td>
<td>57</td>
</tr>
</tbody>
</table>
4 FlowPass: Understanding password misuse in Android 105

4.1 Introduction ................................................. 105

4.2 Problem Statement ......................................... 108
  4.2.1 Threat model ........................................... 108
  4.2.2 Categorizing password misuse ....................... 109
  4.2.3 Secure password usage ................................ 112
  4.2.4 Challenges. ............................................. 112
  4.2.5 Design goals ........................................... 113

4.3 Analyzing the Design Space ......................... 114
  4.3.1 General-purpose static analysis .................... 114
  4.3.2 Specialized static analysis .......................... 115
  4.3.3 General-purpose dynamic analysis ................. 116
  4.3.4 Specialized dynamic analysis ...................... 118
  4.3.5 Summary ............................................... 118

4.4 FlowPass: Design and Implementation ............ 119
  4.4.1 Observations and design decisions ............... 119
  4.4.2 Design overview ...................................... 120
  4.4.3 Storing taint tags ................................... 122
  4.4.4 Propagating taints .................................. 124
  4.4.5 Sources and sinks .................................. 126
  4.4.6 Provenance tracing .................................. 127
  4.4.7 The semi-automated workflow .................... 128

4.5 Evaluation ................................................ 130
  4.5.1 Methodology .......................................... 130
  4.5.2 How effective is FlowPass? ......................... 131
5 Conclusion

Bibliography
Illustrations

2.1 TLS handshake and secrets. ................................................. 10
2.2 Android application lifecycle. ........................................... 13
2.3 Android JSSE architecture. .............................................. 14
2.4 The overview of the analysis framework. ............................ 16
2.5 The experiment scenario. .................................................. 20
2.6 Relationship between HTTPS and TLS implementations on Android. . 26
2.7 SSL_SESSION_free function in BoringSSL. ............................ 28
2.8 Excerpts of session management code from Conscrypt on Android 7.1. 32
2.9 SSL_SESSION structure on Android 7.1.2. .............................. 38
2.10 Measuring secret retention after accessing popular sites. .............. 39
2.11 Rendering TLS communication successfully. .......................... 42
2.12 Clearing sessions by hooking Android Activity. ........................ 44

3.1 Authentication steps in client applications. ............................ 58
3.2 The dataflow of a user password input on Android. .................. 70
3.3 Authentication steps when unlocking a device. ........................ 81
3.4 The password usage patterns in the studied apps, as well as the pattern and
capsulation that KeyExporter achieves. ................................. 86
3.5 Integration with KeyExporter is easy. .................................... 90
3.6 Integrating passwdSafe with KeyExporter. ............................ 96

4.1 Motivating examples. ....................................................... 111
4.2 The architecture of FlowPass. Dashed boxes represent our modification or addition to Android Open Source Project (AOSP). We have further labeled the boxes as sink, source, taint storage, or provenance tracing. Dashed boxes without labels are modified for taint propagation. 121

4.3 Structure of String and Array implementations in Android. 123

4.4 An example provenance trace. 127

4.5 The semi-automated workflow of FlowPass. 129

4.6 A provenance trace that shows how password leaks in a popular app, Dark Horse. 134

4.7 Logcat dump in DoorDash after trying to log in with fake credentials. 137

4.8 Experiment with Chase bank. 138

4.9 Comparisons between Google Play and Chinese apps. Since the app stores have different categories, we have merged similar categories for easy comparison. Red bars are vulnerable apps that directly leak passwords to the network or via Logcat. Yellow bars are apps that send passwords to TLS. Blue bars are the apps that encrypt passwords. The remaining grey bars are the apps that our framework did not fully cover. 139
Tables

2.1 Surviving master keys under various configurations. .......................... 22

3.1 Exposure of passwords in application memory. ................................. 65
3.2 Results for the tested keyboard applications. ...................................... 72
3.3 Results for the tested sample apps. .................................................. 76
3.4 Our proposed fixes, SecureWidget and KeyExporter, can successfully address the password retention problem. A’ is the version of app A running on Android framework with SecureWidget; A† is the version of app A integrated with KeyExporter and SecureWidget. Since Yelp is not open source, we were only able to apply SecureWidget, but not KeyExporter. 93

4.1 Taint propagation logic. Variables \(a\) and \(b\) are strings or arrays. Variable \(c\) can be any type; but if it is not a string or array, it does not have a taint tag, i.e., \(\tau(c) = \emptyset\) ................................................................. 125
4.2 An example instrumentation for taint support. ................................. 125
4.3 High-level statistics. ................................................................. 135
Chapter 1

Introduction

1.1 My Thesis

Today, mobile phones are part of everyday life, completely changing the way people think and behave. Mobile usage has been exceeding desktop usage as more people surf the web, shop online, and use social media with smartphones [1, 2, 3]. Also, the data shows that the trend will continue; the number of smartphone users expects to continuously grow from 5.1 billion in 2018 to around 7.2 billion in 2021 [4], dominated by the Android operating system (85 %) [5].

At the same time, this accelerated growth has brought with it unprecedented new threats related to user privacy, making mobile phones interesting targets for hackers. Firstly, smartphones contain a lot of user sensitive data. Also, in comparison to PC, smartphones have various attack vectors. For example, mobile devices are often connected to public WiFi, and a malicious eavesdropper can easily intercept communications. Also, attackers have chances to gain physical access to a device by stealing or taking lost phones, which allows them to access sensitive data stored on the phones.

However, it is challenging to protect sensitive data on mobile phones because mobile operating systems are very complex. For example, the Android operating system, which is mostly used in the world [5], consists of various open-source projects, and additional software layers such as Java Runtime (ART) on the top of Linux operating system. That complexity makes it difficult to implement the necessary security features safely. Moreover,
there are myriads of apps released, and many of them handle user sensitive data directly or indirectly. However, since developers have varying levels of secure coding practice, apps may misuse the sensitive data stored on the phone or obtained from the sensors. Indeed, recent studies have repeatedly found that app developers implement security features poorly. For example, apps have been reported to misuse TLS libraries [6, 7, 8], cryptographic APIs [9], OAuth protocols [10], and fingerprint APIs [11], risking the user’s privacy.

To make things worse, attack techniques have been more sophisticated. For example, these days, even sensitive data in volatile storage has become a target of adversaries. With memory disclosure attacks, an unprivileged attacker can steal sensitive data from device memory. These attacks frequently make the headlines: recent vulnerabilities that could lead to such attacks include HeartBleed [12], Meltdown [13], and Spectre [14]. Also, if adversaries can gain physical access to a device, they may be able to directly dump its memory, e.g., via a “cold boot” attack [15, 16], or even through its USB connection [17]. Memory disclosure attacks pose a serious threat, as sensitive data, such as cryptographic private keys and passwords, is easily reused if stolen (see, e.g., [18]).

What are the implications for those phenomena, and how can we secure sensitive data on mobile devices? To answers the questions, I have analyzed a wide range of applications and the Android operating system. My analysis has discovered various patterns of mistakes in the Android framework and popular apps that can be exploited by adversaries, which confirms the thesis that developers repeat mistakes in managing sensitive data, which, in turn, risks user privacy in the mobile ecosystem. To mitigate this issue, I have tried various approaches, and my dissertation concludes that it is possible to reduce developers’ mistakes in practice by abstracting critical routines in the operating system level and providing a set of APIs, and by detecting misuse at the early stage of development with lightweight dynamic analysis.
1.2 Contribution and Outline

I have used various analysis techniques to find developers’ mistakes in popular apps and demonstrate that a practical attack is possible to exploit them. My research result has had a practical impact on the industry by finding various previous unknown bugs, and reporting them to vendors, including Google. Most vendors have quickly fixed their problems. Also, my research aims to provide practical solutions to reduce developers’ mistakes. For example, our lightweight taint tracking approach is efficient and effective for identifying password misuse in complex real-world apps. This dissertation presents information which has appeared in previous publications [62, 85], as well as unpublished material. Below I will briefly introduce each chapter.

1.2.1 Removing secrets from Android’s TLS

Cryptographic libraries that implement Transport Layer Security (TLS) have a responsibility to delete cryptographic keys once they’re no longer in use. Any key that’s left in memory can potentially be recovered through the actions of an attacker, up to and including the physical capture and forensic analysis of a device’s memory. I analyzed the TLS library stack used in recent Android distributions, combining a C language core (BoringSSL) with multiple layers of Java code (Conscrypt, OkHttp, and Java Secure Sockets). I first conducted a black-box analysis of virtual machine images, allowing us to discover keys that might remain recoverable. After identifying several such keys, I subsequently pinpointed undesirable interactions across these layers, where the higher-level use of BoringSSL’s reference counting features, from Java code, prevented BoringSSL from cleaning up its keys. This interaction poses a threat to all Android applications built on standard HTTPS libraries, exposing master secrets to memory disclosure attacks. We found all versions we investigated from Android 4 to the latest Android 8 are vulnerable, showing that
this problem has been long overlooked. The Android Chrome application is proven to be particularly problematic. I suggest modest changes to the Android codebase to mitigate these issues, and have reported these to Google to help them patch the vulnerability in future Android systems. Chapter 2 will describe this work in detail, borrowing from [62].

1.2.2 Total Recall: Persistence of Password in Android

A good security practice for handling sensitive data, such as passwords, is to overwrite the data buffers with zeros once the data is no longer in use. This protects against attackers who gain a snapshot of a device’s physical memory, whether by in-person physical attacks, or by remote attacks like Meltdown and Spectre. I found unnecessary password retention in Android phones by popular apps, secure password management apps, and even the lockscreen system process. I have performed a comprehensive analysis of the Android framework and a variety of apps, and discovered that passwords can survive in a variety of locations, including UI widgets where users enter their passwords, apps that retain passwords rather than exchange them for tokens, old copies not yet reused by garbage collectors, and buffers in keyboard apps. I have developed solutions that successfully fix these problems with modest code changes. I will explain this work in Chapter 3 [85].

1.2.3 FlowPass: Understanding password misuse in Android

Traditional usernames and passwords seem to be in no danger of going away, particularly for apps which use them to authenticate users to remote web services. While Android provides UI support for password entry, after that, it’s all up to the applications. Although a developer could certainly opt to use a suitable encrypted password verification protocol, or delegate its authentication to a federated identity provider, Android does nothing to enforce such good behaviors. In order to understand password misuse in Android apps, I present
FlowPass, a dynamic taint tracking system that is efficient and informative. It adds taint tracking support for key data types commonly used for storing passwords, and propagates taint information at the function call level. I have applied FlowPass to 182 popular Android apps, and identified common patterns of password misuse in these apps, including several banking applications. FlowPass also found 13 previously unknown security bugs in popular apps that have each been installed more than one million times. I have reported these misuses to the app vendors, and most have fixed the bugs shortly afterward. Chapter 4 describes FlowPass in detail.
Chapter 2

Removing Secrets from Android’s TLS

2.1 Introduction

Transport Layer Security (TLS) is the most widely-used cryptographic protocol which provides secure communication between a client and server. Confidentiality and integrity of communications are guaranteed by ephemeral secrets shared during a cryptographic handshake.

However, unless these secrets are deleted properly after a session completes, they reside in memory, and thus become vulnerable to memory disclosure attacks, allowing recorded communications to be subsequently decrypted. Attack vectors vary including physical techniques such as “cold boot attack” [15], which physically extract memory chips, and throughout software exploitations like the OpenSSL Heartbleed vulnerability (CVE-2014-0160) which exposes sensitive data in memory to remote attackers without any privileges.

A variety of tactics are used on TLS to ensure secrets are quickly forgotten. For example, modern TLS cipher suites support perfect forward secrecy (PFS), ensuring that key material which must be saved over a long period cannot be used to decrypt previous sessions that an attacker may have recorded. According to SSL Labs’s monitoring [19], about 89% of HTTPS websites support PFS in August 2017, versus only 46% in October 2013.

PFS is essential to managing long-term key material, but what about short-term session keys? Indeed, many TLS libraries like OpenSSL go to great lengths to do memory zeroization of session keys after a session is complete. However, there are two sources of
back-pressure on this. First, TLS gains significant computational performance by caching the results of expensive public-key operations, allowing for fast “session resumption”; this session data could, if captured, be used to compromise any session derived from it. Second, libraries running above OpenSSL may have their own key retention logic, with their own corresponding bugs.

Several recent studies have featured the recover of cryptographic key material as part of a forensic examination. Taubmann et al. [20] provide a technique to extract master secrets at runtime using virtual machine introspection techniques, and Kambic [21] provides a similar analysis of Windows systems. Pridgen et al. [22] investigate Java TLS implementations, finding key material remaining in memory as a consequence of the JVM’s garbage collector. A copying garbage collector may leave multiple dead copies of a key behind in memory that are not “reachable” as live data, yet are still vulnerable to forensic extraction.

**What about Android?** Android’s cryptographic software stack combines layers implemented in C (BoringSSL, derived from OpenSSL) and in Java (Conscrypt, OkHttp, and Java Secure Sockets), providing ample opportunities for subtle bugs to impact key availability.

Several recent Android vulnerabilities underscore the practicality of memory disclosure attacks. For example, a recent vulnerability in a Broadcom WiFi chipset [23] allowed an attacker to take control of the WiFi chip, using it to conduct arbitrary reads and writes into the main CPU’s memory. And of course, once an attacker has physical access to a device, they may have access to further vulnerabilities that allow memory to be dumped (see, e.g., this Nexus 5X issue [17]).

Of course, users of any smartphone device may connect to the Internet over unencrypted WiFi hotspots, allowing attackers to record their communications. Even WiFi’s WPA2 encryption scheme has vulnerabilities [24], enabling adversaries to eavesdrop on supposedly
safe WiFi systems. If a phone or computer is using TLS for all its connections, then these WiFi issues are less of a concern, since an eavesdropper would still see only encrypted traffic.

Lastly, we also note that Android applications have a complicated lifecycle, where the system will put them to sleep and wake them up again later when necessary. If an application is paused before it might have ordinarily zeroized unnecessary key material, that key material might have an undesirably long lifespan.

We started our research with two hypotheses:

- The combination of multiple software layers that implement the TLS protocol, along with Java’s garbage collection system, provide opportunities for subtle bugs in key management.

- Many Android applications will fail to manage cryptographic key material lifetime alongside the Android application lifecycle.

We hypothesize that these two effects will both result in key material that can and should be zeroized instead being available for extraction.

First, we conduct a black-box security analysis. In order to examine various situations, we constructed a virtual-machine framework that supports physical and logical memory dumping. We can drive TLS connections from Android applications, running in our framework, to an external HTTPS server. Since our external server knows the key material, we can search for these keys, anywhere they may occur, in images captured from our virtual machine.

Next, after finding keys that were still resident in memory, we dive into the structure of Android’s cryptography stack, identifying problems in the use of BoringSSL’s reference counting feature that caused keys to living longer than necessary. We propose mitigations...
and measure their effectiveness.

The rest of this chapter is organized as follows. Section 2.2 gives background on TLS concepts and architecture on Android. Section 2.3 provides our black-box analysis method and design details of our automated framework. In Section 2.4, in-depth analysis results are described in detail. Section 2.5 evaluates how this problem is exploitable for attacks in practice. The solutions to address this issue are discussed in Section 2.6. We introduce related work in Section 2.7 and conclude the chapter in Section 2.8.

2.2 Background

In this section, we provide a brief overview of relevant features of the TLS handshake and the Java Secure Socket Extension (JSSE) on Android.

2.2.1 TLS

TLS Handshake and Secrets

Figure 2.1a abstracts the TLS handshake protocol, described in full detail in its RFC [25]. For now, we focus only on how secrets are created and shared. During a handshake, five artifacts are generated or calculated and shared between a client and a server. 32 bytes of client and server randomness are generated by a client and a server, respectively, and shared as a plain text through “hello” messages. Those values are used to calculate a master secret and a key block. A pre-master secret is then shared throughout the next KeyExchange messages, using RSA or Diffie-Hellman key exchange. In RSA key exchange mode, it is the client’s role to generates a 48-byte random value for the pre-master secret. Then, the client encrypts it using the server’s public key and sends to the server. In the Diffie-Hellman scheme, the pre-master secret is derived by mixing generated DH public-private
(a) Overview of TLS handshake and relation with secrets.

(b) Minimum effective lifetime of secrets.

Figure 2.1: TLS handshake and secrets.
pairs with the peer’s DH public value. Whether using RSA or Diffie-Hellman, both sides end up sharing the pre-master secret, while an eavesdropper cannot derive this value.

The premaster secret is significant because all other secrets are derived from it, and thus if it is compromised, confidentiality and integrity on TLS are broken. We note that the Diffie-Hellman construction has perfect forward secrecy (PFS) while the RSA construction does not. This implies that a compromise of the pre-master secret, when used in RSA mode, could be used to compromise older recorded circuits.

Once the pre-master secret is shared, both sides derive a master secret from the pre-master secret, as well as the client random and server random values, using a cryptographic hash function. The master secret is then used to generate a key block together with the client random and server random values, containing all the key material used to protect the confidentiality and integrity of the subsequent network session.

Session Resumption

TLS provides the notion of “session resumption” which allows an abbreviated handshake, avoiding the computationally expensive RSA or Diffie-Hellman handshakes, and reusing the previously shared master secret. This necessarily implies that the master secret must survive beyond the lifetime of any one network connection in order to be reused by a subsequent connection.

Session resumption was supported from the very beginning of SSL through session identifiers [25] and later through session tickets [26]. With session IDs, both the client and server maintain copies of their master key. The client presents the appropriate session ID, and the server can accept or reject that ID. With the newer session tickets, the server no longer needs to preserve its copy of the master key. Instead, the session ticket, stored on the client, has all the necessary state. The ticket is appropriately encrypted to make it safe
to send in the clear. The benefit of this scheme, particularly in large server clusters, is to simplify the server’s need to preserve and replicate state.

For this chapter, we will be concerned with client-side cryptographic state, including the master key secrets, regardless of whether they’re being referenced through session IDs or session tickets.

**Lifetime of TLS Secrets**

Figure 2.1b summarizes the minimum possible lifetime of secrets generated during the handshake when a TLS session is shared across multiple connections. The master secret has the longest minimum lifetime since everything else is unnecessary once any given connection is closed, while the master secret is reused across connections. RFC 5246 specifically recommends that a master secret be maintained for no longer than 24 hours. In our own investigations, we have observed that modern web servers like Apache and Nginx will expire a master secret after only 5 minutes. This implies that a client can and should delete its master secrets in a similar timeframe.

### 2.2.2 Android application lifecycle

One notable distinction between Android applications and traditional desktop programs is the *Android application lifecycle* shown in Figure 2.2. With a traditional desktop operating system, a user will launch a program, then it is loaded into memory, and it will remain running until the user explicitly closes the program. Android operates differently, reserving the right to kill off an application at any time when system resources are exhausted. Applications can also be “paused” and “resumed” by a user, or even “stopped” and “restarted”. These distinctions matter. A “paused” application might still be visible to the user in a multi-window scenario, while a “stopped” application is no longer seen and it’s a good
practice at this point to close active connections and return resources back to the system.

Under these conditions, it’s easy to imagine how TLS key material that should be deleted might well remain in memory. If an Android application doesn’t explicitly set timers to wake itself up, it could be paused for hours, and its key material would then remain present in memory.

An Android application may have “activities” which are visible to the user and “services” which may continue to operate even after a user doesn’t see the activity on their screen. For this research, we focused on Android activities, but we note that services will only make the problem worse, potentially keeping key material in memory long after the activity that used the key material was destroyed.
Android Application

.socketFactory sf = SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket) sf.createSocket("gmail.com", 443);
SSLSession s = socket.getSession();
// ... use socket ...
socket.close();

Android JSSE Interface

javax.net

ServerSocketFactory
SocketFactory

javax.net.ssl

SSLSocketFactory
SSLSocket
SSLSession
 SSLContext

SSL/TLS JSSE Provider

BoringSSL

SSL *SSL_new(SSL_CTX *ctx);
void SSL_free(SSL *ssl);
...

Conscrypt

OpenSSLsocketFactoryImpl
OpenSSLSocketImp1
...

HTTPS JSSE Provider

OkHttp

OkHttpClnet
HttpConnection
...

Figure 2.3: Android JSSE architecture.
2.2.3 TLS Implementation on Android

Android provides HTTPS and TLS implementations through the Java Secure Socket Extension (JSSE) API. The JSSE model is based on a “provider” architecture, providing implementation independence and algorithm extensibility. Figure 2.3 sketches the full stack of JSSE components on recent versions of Android.

A normal developer will use the JSSE APIs available in the `javax.net` and `javax.net.ssl` packages such as `HttpsURLConnection` class. These calls are then delegated to OkHttp, which provides functionality for speaking the HTTP and HTTPS protocols, and Conscrypt, which is a Java-layer wrapper around BoringSSL. BoringSSL itself is a fork of OpenSSL, meant to meet Google’s needs while removing unnecessary functionality. BoringSSL is implemented in C, while the rest of the stack is Java.

2.3 Black-box security analysis

In this section, we describe how we conducted our “black-box” experiments and present our initial findings.

2.3.1 Methodology

Our goal is to be able to discover TLS-related key material, wherever it may be in an Android client’s memory, regardless of issues like user versus kernel memory, whether a given process is still active, or whether the data is live and reachable, or garbage awaiting collection and reuse. If it’s there, in any fashion, we want to find it.

Our black-box analysis framework, outlined in Figure 2.4, consists of Android devices with memory acquisition features, an HTTPS web server, and a test framework. Our custom Android application makes TLS connections using standard JSSE APIs. We use an
Figure 2.4: The overview of the analysis framework.
instrumented web server which can record all of the key material that it sees. Using a vari-
ety of test scripts, we can run our client and server in many different configurations. Once
complete, we dump the client’s memory and the server’s key material. We built tools to
automatically search through the client memory dumps for this key material.

We refer to this as a “black-box” approach because we make no assumptions about
how the client-side application works. If the keys are anywhere in memory, we’ll find
them, simulating the power available to a forensic analyst with a client memory dump and
recordings of prior TLS sessions.

2.3.2 Test Framework

There have been several studies on Android, comparably dumping memory to look for
sensitive data [27, 28, 29, 30, 31]. However, all of these approaches required a manual
approach to capture and search the relevant memory images. The problem with manual
approaches is that they don’t easily scale to examining hundreds of application runs. For
example, Apostolopoulos et al. [27] spent six months examining thirty Android applica-
tions. Furthermore, if the effects being measures are probabilistic in nature, multiple runs
will be necessary, further burdening the data collection and analysis process. Ntantogian
et al. [30], for example, studied application lifecycle issues with credential usage, taking
three months to examine 390 test-cases for thirteen applications.

Our test framework, for contrast, is scriptable and automated. Our system is compara-
bile to the standard Android MonkeyRunner tool, normally meant for bug testing, only with
memory dumping features and with a connection to our instrumented web server for cap-
turing the relevant key material. We can easily run repeated experiments, with or without
varying the experimental parameters.

Of course, our test framework is a special-purpose design, meant only to run our TLS
client app and extract memory images. It would not be suitable for examining general-purpose Android applications for vulnerabilities, although we will see later how we used our system to examine the closed-source Android Chrome web browser. Despite these limitations, we note that most Android apps will use standard Android APIs for their cryptographic communications, so any issues we find here should generalize to any apps using the same APIs.

### 2.3.3 Supporting memory dump on Android devices

Our threat model assumes memory disclosure attacks, and we need to set up that situation on a real Android device as well as an emulator. Android devices do not provide memory dumping as a native feature, requiring us to rebuild the Android kernel to include the LiME kernel module [32], giving the necessary functionality. For contrast, when running Android under the QEMU virtual machine emulator, we can use the `pmemsave` command which does exactly what we need without requiring a custom kernel.

Of course, a raw memory dump doesn’t give an easy view of a process’s virtual address space. To help with this, we created a native module called `pmdump` to extract data from any virtual address space. This allowed for easily scripted queries against a running virtual Android image.

### 2.3.4 HTTPS client on Android devices

We built simple HTTPS / TLS clients for Android to exercise the standard cryptographic libraries. Recent research [33] shows 84% of applications use standard libraries for TLS; our approach allows us to most efficiently exercise the standard Android’s HTTPS / TLS software stack.

We enhanced our simple client to vary the number of concurrent connections and the
degree of memory pressure. We can vary the number of threads and vary how often we might explicitly ask the garbage collector to run. Also, we can use the JSSE libraries in their default manner, resuming sessions whenever possible, or we can explicitly use a fresh SSLContext for each connection, forcing a full public-key handshake, and thus generating many more master secrets.

2.3.5 Server

Our server runs Ubuntu 14.04 with the Nginx web server and its default HTTPS configuration. This supports session resumption with session tickets.

The only necessary customizations on our server are to log all of the cryptographic keys used in the OpenSSL library. As with our customized Android environment, this web server would be unsuitable for use in a production environment, since a log of every cryptographic key used would, in a production environment, represent an unacceptable security vulnerability. After all, the whole point is to forget unnecessary key material! Nonetheless, we want to capture these keys so we know what to search for in our client memory dumps.

2.3.6 Experiment

Figure 2.5 shows our main experimental scenario. For Android devices, we used a Nexus 5 equipped with Android 6, and Android emulators running four different Android versions, from Android 4.3 to Android 7. We varied the number of threads from 10 to 200 on the client, with each making 10 to 500 HTTPS connections to our server. We capped the total number of connections to 5000 per run; beyond this, we managed to crash or freeze our device.

We also varied whether we used the high-level HTTPS APIs versus the lower-level
Figure 2.5: The experiment scenario.
SSL/TLS APIs, to see if this made a difference, helping us ultimately identify which layers might be most responsible for problems. We also ran a series of experiments with Android’s Chrome browser rather than our test application, to see whether the additional complexity of Chrome’s internal layers might make a difference.

Overall, this round of experiments spanned three weeks of effort to capture more than 200 different test cases, with a fair bit of our time spent ironing out bugs in our framework versus bugs in Android itself. We contrast this with Ntantogian et al. [30], who spent three months to evaluate 390 cases for 13 applications.

### 2.3.7 Results

Our initial results show that Android is effective at removing pre-master secrets and session keys but is not effective at removing master secrets. We observed this when we configured our experiments to perform regular session reuse as well as when we configured them to do new public-key operations on every session. We observed these issues regardless of Android version or other experimental parameters.

Table 2.1a shows the remaining master secrets when varying the number of threads and connections. These results are quite similar across the various emulator configurations, regardless of the emulated Android version or the number of connections. For contrast, our Nexus 5 phone shows significantly more remaining master keys. We concluded that this difference is due to device configuration (memory size, etc.), which makes events like garbage collection happen less frequently in the real hardware. There is no fundamental difference in the logical behavior.

We also note that none of the remaining master secrets that we found were duplicates.

---

*Creating a new SSLContext for each connection, so a full handshake on every connection.
†Using the default SSLContext to support the session resumption.
<table>
<thead>
<tr>
<th>Threads</th>
<th>Conn.</th>
<th>Total Conn.</th>
<th>Nexus</th>
<th>Emul 4.3</th>
<th>Emul 5</th>
<th>Emul 6</th>
<th>Emul 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>100</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>500</td>
<td>29</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>16</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>500</td>
<td>5000</td>
<td>11</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>500</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>1000</td>
<td>12</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>200</td>
<td>10</td>
<td>2000</td>
<td>85</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

(a) Baseline measurements, varying the degree of concurrency*.

<table>
<thead>
<tr>
<th>Devices</th>
<th>Nothing</th>
<th>GC</th>
<th>BG</th>
<th>Kill</th>
<th>GC → BG</th>
<th>GC → Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nexus</td>
<td>29</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Emul 4.3</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Emul 5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Emul 6</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Emul 7</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

(b) After 500 connections and other additional actions*.

<table>
<thead>
<tr>
<th>Application</th>
<th>Nothing</th>
<th>BG</th>
<th>Kill</th>
<th>1hour → BG</th>
<th>BG → 1hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPS App*</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>TLS App*</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(c) HTTPS APIs vs. TLS APIs

<table>
<thead>
<tr>
<th>Application</th>
<th>Nothing</th>
<th>BG</th>
<th>Kill</th>
<th>1hour → BG</th>
<th>BG → 1hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPS App†</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TLS App†</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chrome</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(d) Android Chrome vs. our test application

Table 2.1: Surviving master keys under various configurations.
Each one represents a distinct key. This suggests that we don’t have a problem originating from Java’s copying garbage collector or careless object copying.

For our next experiment, we kept the number of connections constant at 500, instead of varying the treatment of the Android system after the connections are complete. Our experiments considered explicitly invoking the system’s garbage collector (“GC”), placing our test app into the background (“BG”), killing our test app (“Kill”), as well as combinations of these actions.

Table 2.1b shows the results across different emulated versions of Android as well as our Nexus 5 device. Several things jump out. First, once we conduct any sort of post-connection action, this immediately normalizes the difference caused by device configuration between our phone and our emulator.

Additionally, we note that even killing off our application does not eliminate its keys from memory. Since the OS process is dead, this suggests that Android is lazy about zeroing out such memory. Of course, memory is a scarce resource in a mobile phone and it will eventually be reused and cleared, but our results suggest the kernel could zero out dead process memory more aggressively.

Short of killing a process, however, we continue to find unused master keys in memory, regardless of the version of Android in consideration, with the total number being remarkably constant across different versions and through different scenarios. Clearly, these keys are not being held alive through an absence of execution of the garbage collector or through any actions that happen as the Android app moves into a paused state in its lifecycle.

Table 2.1c shows the result varying the APIs in use by our test application: HTTPS vs. TLS, and explicitly include a one-hour wait after establishing consecutive 20 connections, to see if any timer-related activity might kick in and erase unused keys. From Table 2.1c, we can observe that the HTTPS layer is somehow responsible for keeping our master keys
alive even though they are never needed, as we are instead forcing a full TLS handshake on every connection.

Table 2.1d includes a measurement taken against the Android Chrome application. We build a web client that behaves similarly to our Android native client, including making a similar number of concurrent connections to the same server. We see one master secret generated at the start and successful reuse of it for subsequent connections. Even after a one hour wait, the master secret remains in memory for both the native test app and Chrome.

2.3.8 Observation and Raised Questions

These experiments reveal problems in how Android’s HTTPS/TLS stack manages its master secrets. Considering that Android properly delete pre-master secrets and session keys, clearly, its developers intended to pay careful attention to key material lifetimes.

Nonetheless, we’re left with a number of questions that we can only answer by digging into the Android code. Our black-box analysis has helpfully removed some issues from consideration, but remaining questions include:

*Question 1* Is master secret retention the result of bugs or a deliberate performance vs. security trade-off?

*Question 2* Why does varying the number of concurrent connections have no impact on the number of remaining master secrets?

*Question 3* Why do explicit calls to the garbage collector and moving an application to the background have the same effect, while both fail to remove every secret?

*Question 4* Why does the HTTPS API have different secret-saving behavior than the lower-level TLS APIs?

*Question 5* Why do secrets last longer when using the default SSLContext rather than when creating new contexts for each connection?
2.4 In-depth Analysis of Android Framework

Following the questions that our black-box approach raised, we proceeded to analyze the Android codebase, in particular, looking at BoringSSL, Conscrypt, and OkHttp. These are still quite substantial in size, so we needed an approach to narrow our understanding of how they worked. Our main approach was to annotate these libraries with logging calls, allowing us to see the order in which they perform their various operations, which we can then study afterwards.

In particular, we need to know every time that BoringSSL allocates and frees memory that contains cryptographic state. We did this by hooking the object creation and deletion events as well as the underlying memory allocation events (i.e., `OPENSSL_malloc()` and `OPENSSL_free()`). We similarly added logging to Android Java code for Conscrypt and OkHttp, allowing us to correlate events at the Java level with events in the C level.

In the following subsections, we go into more detail of how OkHttp, Conscrypt, and BoringSSL interact with each other and how they manage memory. To be clear, we couldn’t have written this without the logging and tracing that ultimately helped us see how the layers interact.

2.4.1 Overview

We summarize the relation among the three modules in Figure 2.6. There are three key concepts of TLS implementations that are important to understand: Context, Connection, and Session. A context encapsulates the TLS implementation itself, including the implementation’s configuration and supported cipher suites. A context is typically shared across every TLS connection. Importantly, contexts are responsible for caching session state for fast session resumption. A connection represents a single TLS connection, typically corresponding to a specific TCP/IP socket session. Every connection has a corresponding
session and context within which it’s defined and which manages its long-term state. Once the TCP/IP socket closes, the connection is done. Conversely, a session represents the TLS state that may be resumed in an abbreviated handshake. Sessions are also associated with contexts. When a TCP/IP socket closes, the session state, as stored in the context, will survive.

Those three concepts are supported by the SSLContext, SSLSocket, and SSLSession classes in the javax.net.ssl package in JSSE. Additionally, SSLSocketFactory class is provided by JSSE that encapsulates the SSLContext class, providing a convenient wrapper for creating many sockets without having to juggle the SSLContext objects. Since the javax.net.ssl package only contains interfaces, corresponding concrete classes exist both in Conscrypt and BoringSSL as shown in Figure 2.6. For the SSLContext class, the OpenSSLContextImpl and SSLParametersImpl classes in Conscrypt provide the glue routines between Java and C, and the SSL_CTX structure in BoringSSL contains the actual TLS context.
Conscrypt’s SSLParametersImpl class is an important class because it contains the ClientSessionContext and ServerSessionContext classes which are responsible for tracking previous sessions. In turn, Conscrypt’s OpenSSLSocketImpl class corresponds to the SSL structure in BoringSSL, supporting JSSE’s interfaces for SSLSocket. And likewise, Conscrypt’s OpenSSLSessionImpl corresponds to BoringSSL’s SSL_SESSION structure, which underpins JSSE’s SSLSession class.

Things get interesting when you track where references are stored. For example, OkHttp maintains a “connection pool”, which in turn holds recent Connection objects for later reuse, allowing support for HTTP/2.0’s multiplexing features. (The API client can continue to operate as if it were opening a fresh TCP/IP socket for each HTTPS request, while the implementation is free to multiplex these requests over the same socket.) Consequently, OkHttp’s connection pool will keep around Conscrypt connections and sessions, even though the connections may be inactive, and those will, in turn, keep alive their corresponding structures in BoringSSL.

We now know why we never find duplicate master secrets in memory. This is because those master secrets are stored in BoringSSL’s SSL_SESSION structure. The Conscrypt and OkHttp layers keep references to these BoringSSL structures, but never directly copy them. In the subsections below, we will consider each of these modules, in turn, and describe how their interactions ultimately created the master-secret retention problems that we observed.

2.4.2 BoringSSL

BoringSSL provides the full functionality of TLS by itself, and can be used by C programs without interacting with any of the Java modules. Subroutines for handshaking and managing secrets are all implemented in BoringSSL. The pre-master secret is deleted properly after finishing handshake. Every master secret is allocated and located within
an SSL_SESSION object after establishing a session, and it is cleaned when no other object retains a reference to this SSL_SESSION object. Because BoringSSL is implemented in C, the programming language runtime system provides no assistance in detecting when an SSL_SESSION is dead. Instead, BoringSSL supports manual reference counting for its data structures. All major data structures have a reference count field, and clients of BoringSSL are responsible for issuing calls to increment or decrement these counts. In typical usage, however, the counter is initialized to one when the data structure is created, and each free-function (e.g., SSL_SESSION_free()), will decrement the reference counter. If the free-function finds the reference counter is zero, it will then “cleanse” (i.e., zeroize) the memory region before handing the memory back to the memory allocator. If, however, the reference count is non-zero, then the free-function will only decrement the counter and otherwise do nothing else. This logic is shown in Figure 2.7.

When BoringSSL is used on its own, these reference counts appear to be managed correctly. For example, when an SSL structure is created for a TLS connection, it can be initialized with an existing SSL_SESSION or with null. In the latter case, it creates a new

```c
void SSL_SESSION_free(SSL_SESSION *session) {
    if (session == NULL ||
        !CRYPTO_refcount_dec_and_test_zero( &session->references)){
        return;
    }
    ...
    OPENSSL_cleanse(session->master_key, ...);
    ...
}
```

Figure 2.7 : SSL_SESSION_free function in BoringSSL.
SSL_SESSION, with a reference count of 1 and performs the full TLS handshake. In the former case, it increments the reference count on the SSL_SESSION with which it has now been associated.

One interesting complexity occurs in server mode. The SSL structure is used for both client and server communications. In server mode, after the handshake is successfully finished, a reference to the SSL_SESSION structure is stored in the SSL_CTX structure, increasing its reference count by one more. This means that the SSL_SESSION object is not deleted when its parent SSL object dies and decrease its reference count. In client mode, no such caching action happens. That means BoringSSL does not, by itself, support session resumption in the client mode. To work around this, another layer has to manage the session state by referring to it from a safe place and increasing its reference count whenever the session is established. This action is performed by Conscrypt, in Java. If we create a client socket using Conscrypt, SSL_SESSION’s reference count becomes two.

Our tentative conclusion is that BoringSSL has no particular bugs in its manual reference counting, but we decided to dig deeper into Conscrypt’s use of the BoringSSL reference counting feature.

2.4.3 Conscrypt

Unlike BoringSSL, Conscrypt does not work alone; its whole purpose is to present an analog of the BoringSSL library to a Java programmer. Conscrypt’s SSLParametersImpl, OpenSSLSocketImpl, and OpenSSLSessionImpl classes are exactly mapped one-to-one with BoringSSL’s SSL_CTX, SSL, and SSL_SESSION structures, and they have the same lifetimes. For example, when SSLParametersImpl is created, it calls the initialization routine of SSL_CTX on BoringSSL. The SSLParametersImpl object, in Java, maintains a C pointer to the BoringSSL SSL_CTX, stored as an integer field in a Java object. Needless
to say, this means that Conscrypt must be very sensitive to the correct use of the reference counting APIs of BoringSSL.

As mentioned above, Conscrypt implements the client-side session resumption functionality. To accomplish this, it maintains a session cache in the `ClientSessionContext` object. When a TLS handshake is about to start, an `SSLSocket` object is created, and it queries its parent `SSLContext` if there is a previous session with the same host and port. In turn, `SSLParametersImpl`, the concrete class of `SSLContext`, searches the previous `SSLSession` object from the session cache on its child `ClientSessionContext` object. If there is a previous session, the `SSLSocket` object copies the session object in it and calls its corresponding function in BoringSSL to set the current session to the found `SSL_SESSION`. Then, the abbreviated handshake happens.

If there is no previous session in the session cache, a new `OpenSSLSessionImpl` is created, and it does the full handshake. After the handshake, the new session is stored in the session cache in `ClientSessionContext` and also it increases the reference count of `SSL_SESSION` on BoringSSL one more, making its reference count two. When a socket is closed, `SSLSession`’s `free()` is called, and its reference count decreased from two to one. Since the reference count is still not zero, it is not removed and can be reused later in session resumption.

Conscrypt’s session cache supports TLS session resumption, but we found it creates problems in releasing keys. Figure 2.8 includes three problematic codes that contribute to keeping master secrets unnecessary long in the memory without any performance benefit. (This answers Question 1 in Section 2.3.8. Secret retention seems to be the result of a bug, not a performance trade-off.)
public class OpenSSLSessionImpl implements SSLSession {
    ...
    @Override
    protected void finalize() throws Throwable {
        try {
            if (sslSessionNativePointer != 0) {
                NativeCrypto.SSL_SESSION_free(sslSessionNativePointer);
            }
        } finally {
            super.finalize();
        }
    }
}

abstract class AbstractSessionContext implements SSLSessionContext {
    ...
    private static final int DEFAULT_SESSION_TIMEOUT_SECONDS = 8 * 60 * 60;
    ...
    private final Map<ByteArray, SSLSession> sessions
        = new LinkedHashMap<ByteArray, SSLSession>() {
            @Override
            protected boolean removeEldestEntry(
                Map.Entry<ByteArray, SSLSession> eldest) {
                boolean remove = maximumSize > 0 && size() > maximumSize;
                if (remove) {
                    remove(eldest.getKey());
                    sessionRemoved(eldest.getValue());
                }
                return false;
            }
        };
    ...
}

(a) finalize() method on OpenSSLSessionImpl.

(b) Removal routine in AbstractSessionContext.
public class OpenSSLContextImpl extends SSLContextSpi {

... 

private static DefaultSSLContextImpl DEFAULT_SSL_CONTEXT_IMPL;
...

protected OpenSSLContextImpl() throws GeneralSecurityException, IOException {
    synchronized (DefaultSSLContextImpl.class) {
        this.algorithms = null;
        if (DEFAULT_SSL_CONTEXT_IMPL == null) {
            clientSessionContext = new ClientSessionContext();
            serverSessionContext = new ServerSessionContext();
            DEFAULT_SSL_CONTEXT_IMPL = (DefaultSSLContextImpl) this;
        } else {

        }
    }
    
    
}

(c) The default constructor in OpenSSLContextImpl.

Figure 2.8: Excerpts of session management code from Conscrypt on Android 7.1.

Depending on GC

One of the main causes of master secret retention is that Conscrypt places critical code in its finalize() method as shown in Figure 2.8a. In Java, the finalize() method is only invoked when garbage collection is triggered due to insufficient heap memory in JVM. This means it can take quite a while, after an OpenSSLSessionImpl becomes garbage before its finalize() method might be called. This, in turn, means that Conscrypt will keep the underlying BoringSSL session state, including the master secret, live well beyond when it should have been cleansed.

To make matters more complicated, a Java programmer might maintain a live reference
even though it didn’t intend to. Memory leaks are certainly a well-understood issue in Java or any other programming language that relies on garbage collection. This means that the `finalize()` method here might never get called. Even a well-intentioned Java programmer might deliberately keep a reference to an `SSLContext` which will, in turn, keep references to `SSLSession` objects. This issue appears to explain why master keys survive even after explicit calls to the garbage collection.

**LRU implementation of the session cache**

Conscrypt maintains the session caches holding `SSLSession` objects for session resumption in its `ClientSessionContext` and `ServerSessionContext` classes. The ideal deletion logic would be to remove session objects after a designated time, such as ten minutes or perhaps one hour. Unfortunately, those classes do not provide such an explicit deletion routine. Instead, as the code in Figure 2.8b shows, the session cache is designed to work in an LRU fashion (see the code dealing with `removeEldestEntry()`). Removing the eldest `SSLSession` only happens when a new `SSLSession` is added to the session cache, and the session cache is full. Thus, even though the master secrets may have expired, they will only be removed as a result of the LRU cache’s eviction logic. This lazy deletion allows attackers to get as many master secrets as there exist slots in the session cache; the default size of the session cache is 10 and 100 for a client and a server, respectively.

In our previous experiment (see Table 2.1d), when we use our Android client supporting session resumption, we always found one master secret, regardless of what actions we might have taken to try to force it to clean up, including an hour of waiting. That is because the master secret is in the session cache, and it is never be removed since the session cache is never full. If we access more than ten sites, old sessions will finally be allowed to expire, but new ones will continue to hold master secrets alive. Furthermore, another problem
in Figure 2.8b concerns the default session timeout of 28,800 seconds (8 hours) which is unnecessarily long. Additionally, this value is only used to check if the session is valid. This timeout is not used to remove sessions from the session cache.

Singleton SSLContext

Most of the root classes in Figure 2.6 are created using a “singleton” pattern and managed globally, so there is only ever one instance of these Java classes. This prevents SSLSession objects from ever being garbage collected. This issue also applies to the ConfigAwareConnectionPool, SSLContext, SSLSocketFactory, and SSLParametersImpl classes. The code in Figure 2.8c shows that the default SSLContext member is defined as private and static, and thus it is initialized once when SSLContext is first created with the default mode, and it will never become garbage. Consequently, while an application is running, all master secrets relating to the default SSLContext will never be removed. (This answers Question 5 in Section 2.3.8. The default SSLContext never becomes garbage.)

2.4.4 OkHttp

The OkHttp library provides HttpsURLConnectionImpl as the concrete class for the JSSE HttpsURLConnection interface. Also, it provides a ConnectionPool class, internally maintaining a cache of connections. This connection pool also creates issues with the retention of master secrets, because ConnectionPool stores Connection objects which hold SSLSocket internally. When an HTTPS connection is established, an internal SSLSocket is created and the Connection object holds it. Adding and removing a connection from the connection pool is performed automatically, so there is no way for the developers to influence the extent to which the connection pool impacts master secret retention.

However, unlike Conscrypt, ConnectionPool implements eager deletion of connec-
tions using a Timer. Therefore, it ensures that Connection objects are deleted after a 5-minute timeout, corresponding nicely to the default expiration policies used by Apache and Nginx. This default timeout is good, but in order to delete master secrets, the garbage collector must also run after Connection objects are cleared in the pool. This additional requirement still causes an undesirable situation.

Consider the scenario where an app is interacting with an HTTPS server. Before the five-minute timer expires, the user moves the app to the background, perhaps to answer a phone call. Unfortunately, in this case, the master secrets will not be removed because the application is no longer active. The garbage collection will never get called unless the application is woken again by the user. Inactive applications could potentially set a timer to wake up and run the garbage collector, but this is not a default behavior. (This answers Question 3 in Section 2.3.8. Background apps are never garbage collected.)

Our previous experiment (Table 2.1c) shows this worst case in action. One hour after moving the application to the background, the master secrets are still found in memory for an app using the HTTPS APIs. For contrast, the master secrets are all removed when the application instead used the lower-level TLS APIs. This is because GC is usually called at the moment when the application is going to the background. The lower-level TLS API does not have the connection-pool structure, so it won’t hold onto as much key material. (This answers Question 4 in Section 2.3.8. The higher-level APIs have a connection-pooling mechanism that keeps key material alive.)

2.4.5 Summary of the problem

BoringSSL itself correctly supports reference counts to track access to its internal structures, and will properly zeroize key material once the reference count goes to zero. OkHttp has a timer to detect expired entries in its connection pool. The biggest issue, however, is
how all these layers interact with Conscrypt’s thin Java wrapper around the BoringSSL library. In Conscrypt, cleanup depends on garbage collection to run, allowing master secrets to survive in memory long after they should be removed. Consequently, the number of master secrets in memory is constant—the size of the session cache. (This answers Question 2 in Section 2.3.8. The connection pool’s size does not vary with the number of concurrent connections.)

- Using `HttpsURLConnection` with the default `SSLContext`: At least ten master secrets can be found during the whole application lifecycle once the application accesses more than ten sites.

- Using `HttpsURLConnection` with a disposable `SSLContext` (i.e., with session resumption not supported): At least five secrets can be found once the application accesses more than five sites until five minutes elapse and then GC is called. If the application is paused before those five minutes expire, the secrets will persist since there is no chance to call GC.

- Using `SSLSocket` with the default `SSLContext`: At least ten master secrets can be found during the whole application lifecycle once the application accesses more than ten sites.

- Using `SSLSocket` with a disposable `SSLContext` (i.e., with session resumption not supported): Some secrets remain after connections are completed, but they are deleted quickly when the application is paused or sent to the background.

### 2.5 Evaluation of Attack Feasibility

In this section, we evaluate the impact of the master secret retention problem. We discuss additional conditions required to make this issue practical, show they are realistically ex-
ploitable, and demonstrate that attackers can recover the plaintext from recorded HTTPS
sessions using our tools.

2.5.1 Threat model

We have two assumptions in our threat model. First, the attacker is able to passively capture
network packets. This may occur over WiFi or any other network, and is quite feasible
in practice. One recent study showed that packet capture from WPA2-encrypted WiFi is
feasible [24], and of course, the attacker may control the network infrastructure.

We do not assume the attacker begins with any compromised private keys. As such, our
attackers cannot, at the start, successfully decrypt the TLS sessions between the Android
phone and its remote server counterpart.

Also, we do assume that attackers possess Android memory disclosure vulnerabilities,
which can be physical memory dumping exploits such as the vulnerability in the Nexus
5X [17]. They also can be software-based exploits that allow them to access contents in
memory remotely, such as the recent vulnerabilities in WiFi chipsets [23] and Bluetooth
chipsets*. 

Given these assumptions, it’s valuable to consider how realistic these attacks might be
in practice. In particular, two conditions should be satisfied.

- The 48 bytes corresponding to a master secret must be extracted from the device’s
  memory in a reasonable time.

- Master secrets must remain in the phone, while the phone is being actively used,
  rather than simply quiescent. Otherwise, memory needs from active apps will drive
  the Android system to reclaim memory from a quiescent app.

*See https://www.armis.com/blueborne/
struct ssl_session_st {
  CRYPTO_refcount_t references;
  int ssl_version;
  ...
  uint32_t key_exchange_inf;

  int master_key_length;
  unit8_t master_key[SSL_MAX_MASTER_KEY_LENGTH];

  unsigned int session_id_length;
  unit8_t session_id[SSL_MAX_SSL_SESSION_ID_LENGTH];
  ...
}

Figure 2.9: SSL_SESSION structure on Android 7.1.2.

2.5.2 Extracting a master secret

Given a memory image, an attacker must locate the master keys, which could be anywhere in memory. In our earlier experiments, we already know the 48-byte master secrets because our modified HTTPS server logged them for us. In practice, this will not be available.

Luckily for attackers, the surrounding C structure creates patterns that are easy to recognize. Figure 2.9 shows the SSL_SESSION structure on Android 7. The master_key variable is located after ssl_version and master_key_length. Also, the secret is followed by a session_id_length variable. All those variables have well-defined values and they occupy 12 bytes in their specific positions. This signature pattern is sufficient for a rapid search through a memory image. We confirmed that we could extract all master secrets from gigabytes of a memory image in seconds, both from our HTTPS applications and from Android Chrome. (This also suggests that Android Chrome is based on a similar
(a) Result with HTTPS apps after accessing 20 sites.

(b) Result with Chrome after accessing 20 sites.

(c) Result from the long-term experiment with Chrome after accessing 5 sites.

Figure 2.10: Measuring secret retention after accessing popular sites.

BoringSSL codebase.)

2.5.3 Measuring secret retention in active use

To be exploitable, master secrets should reside in memory long enough in a phone being used as real users might operate their phones. In practice, a phone does more than run a single app. Many apps operate background services to download emails, social network streams, and texts. A modern mobile phone is never truly quiescent.

To simulate this, we conducted this additional experiment using our custom HTTPS
client and the Android Chrome application, accessing 20 popular HTTPS websites. After that, our experiments considered four conditions: our application or Chrome running in the background, our application or Chrome being killed, our application or Chrome being forced to share the phone with the YouTube app, or a combination of these effects (killing the app and running the YouTube app). While this was going on, we captured periodic memory dumps to assess the number of master keys present in memory.

The results are shown in Figures 2.10a and 2.10b for our application and Chrome, respectively. The Chrome results are particularly surprising. Note the size of the y-axis. We extracted hundreds of master secrets from Chrome, many minutes past their last use. About ten master secrets are accumulated in memory for each new site, without any deletion. This suggests that Chrome has made a performance-vs-security tradeoff, keeping key material live rather than allowing it to be deleted aggressively.

The only case where Android’s memory management saved us was in the case when we killed the app or Chrome browser and subsequently ran YouTube. YouTube’s memory demands forced Android to reclaim memory for YouTube, zeroing out the expired pages from our app.

Lastly, we conducted a long-term experiment for Chrome giving different events to see whether Chrome eventually cleans up its keys. We accessed five sites using Chrome and then left it in memory. We performed various activities every hour, including rotating the phone, running YouTube briefly, and even letting YouTube run for two days solid. (We chose the YouTube application because it alone creates non-trivial workload on the phone.) The result in Figure 2.10c clearly shows that no such events impact master secret retention in the long inactive application. YouTube alone failed to push the system to kill background applications on today’s powerful devices though it constantly downloads data, decodes video, and displays it on screen.
Our results show that attackers have an unnecessarily large window of time to get master secrets from victims’ phones. While “power user” phones might forcibly reclaim memory, many users don’t use their phones so aggressively, and as a result, their TLS master keys will remain in memory.

2.5.4 Decrypting TLS communication

Our result shows that the master secret retention problem is a real concern and an especially serious one for Android Chrome. It’s entirely feasible to imagine a nation-state adversary, controlling its cellular infrastructure, which can record all network traffic from targeted users. Later, if they are able to access a target user’s phone, whether by an over-the-air hack or by physical interception, they will then be able to decrypt the target’s encrypted communications.

To verify this scenario, we implemented a forensics tool which takes an Android memory image and a captured packet trace as inputs. Then, it extracts all master secrets from the memory image, extracts all TLS streams from the packet trace, and uses the former to decrypt the latter.

Figures 2.11a shows the snapshot of accessing a website using Chrome on the Android 7 emulator. All HTTPS communications attackers see are encrypted as Figures 2.11b. After accessing some sites, we used the phone as a normal user, making calls, and sending text messages. After leaving the phone locked for a day, we dumped the memory, modeling an attacker who physically captured the phone. As Figure 2.11c shows, our tool recovered the victim’s web traffic. Furthermore, since decrypted HTTPS requests will often include usernames and passwords, or HTTP session cookies, such credentials will allow the attacker to impersonate the user to the site.
2.6 Discussion

In the section, we suggest modest changes to mitigate the master key retention problems that we discovered and discuss related issues.

2.6.1 Solutions

The fundamental solution is to resolve the conflict in the object management between Conscrypt and BoringSSL. That is, Conscrypt should be modified to use the reference count feature, deleting SSLSession objects when the reference count is zero. This synchronization of the logic with BoringSSL will remove this issue. Another fundamental solution is to move the session management logic from Conscrypt to BoringSSL, which could handle this internally.

These solutions, however, require invasive changes and might lead to subtle compati-
bility issues. As such, we suggest some simpler alternatives.

**Strawman solution**

The simplest solution is to cleanse master secrets right after a session completes. To remove master secrets promptly, the call to free SSL_SESSION in the `finalize()` method needs to be moved to a location where it will execute eagerly. To test this, we implemented logic like this in the `SSLSocket.close()` method that is always called when a session completes. After applying this simple fix, and rerunning our experiments, no master secrets were found. However, this is not a good solution since it does not support session resumption and thus gives poor performance. Whenever closing the `SSLSocket` object, our strawman solution arranges for the `SSL_SESSION` object in BoringSSL to also be removed, wiping the master secret.

Curiously, the session cache on Conscrypt still has its `SSLSession` object in it. Next time, when trying the access to the same server, this `SSLSession` object is retrieved and associated with new establishing `SSLSocket` object, then it attempts to do the abbreviated handshake. However, the underlying `SSL_SESSION` object has already removed, so BoringSSL makes a new `SSL_SESSION` and is forced to do the full handshake. This straightforward measure is not a practical solution, but it confirms our analysis that the root cause of remaining master secrets is the dependency on GC, which delays their zeroization.

**Hooking Android’s core framework**

A better solution is to modify the Android core library to remove master secrets as part of the Android application lifecycle, ensuring that master secrets are cleaned up, regardless of the lifecycle state of an application. Done this way, our key retention problem can be addressed by triggering a zeroization routine when the application’s lifecycle is changed.
As a prototype solution, we modified the Android Core Framework, hooking the Activity class to trigger the cleansing routine when the application is going to the background or being killed. Figure 2.12 shows our simplified routine applied to the Android framework. It has three steps. First, it removes all sessions in the session cache from singleton objects such as SSLContext and SSLParameters. While the JSSE interface does not explicitly expose an interface to clear the session cache, we found a hack to clear the session cache to use standard JSSE APIs. We clear the cache by resetting the cache size to zero and reverting it, triggering Conscrypt’s lazy deletion routines. Though we delete the session in the

```
Contexts ← ∅
Objs ← GetAllSingleton(OpenSSLSocketFactoryImpl)
Objs ← Objs ∪ GetAllSingleton(OpenSSLContextImpl)
Objs ← Objs ∪ GetAllSingleton(SSLParametersImpl)
for all o ∈ Objs do
    Contexts ← Contexts ∪ getClientContext(o)
end for
for all Ctx ∈ Contexts do
    OrgSize ← Ctx.getSessionCacheSize()
    Ctx.setSessionCacheSize(0)
    Ctx.setSessionCacheSize(OrgSize)
end for
ConfigAwareConnectionPool.getInstance().clear()
System.gc()
```

Figure 2.12: Clearing sessions by hooking Android Activity.
session pools in Conscrypt, OkHttp also has connections in the connection pool that holds an SSLSocket object which in turn has SSLSession.

OkHttp provides ConfigAwareConnection object which is managed on Android as a singleton, so it can be accessed in any place using its getInstance() method. We implemented a clear() method in ConfigAwareConnectionPool class and made use of it from our modified Activity. Though we delete all SSLSession objects from their live parents, cleansing master secrets on BoringSSL is triggered only if the dead SSLSession objects are garbage-collected. Thus, it is important to call System.gc() as the last step. For our tests, we confirmed that garbage collection is always executed when we call System.gc(). After applying this fix, master secrets are properly cleared whenever our test applications are going to the background or being killed.

While our solution appears to work in a relatively non-invasive fashion, the drawback of this solution is adding significant work, including a call to the garbage collection system, exactly when the application is being sent to the background or being killed, which will potentially introduce additional system overhead exactly when a new application is coming to the foreground, which could impact the smoothness of the user experience.

**Concurrent eager deletion**

Our last suggestion is to perform the eager deletion for out-dated SSLSession objects in the session cache. This patch uses a secondary thread, running in the background. The clean-up thread is created when the first SSLSession object is appended to the empty session cache. Once it is triggered, it looks for session cache entries which are expired and deletes them. It otherwise goes to sleep for a suitable amount of time (e.g., one minute), wakes up, and repeats the process. If the session cache is ever completely empty, the thread can terminate, and restart again as necessary.
We believe this is the most effective solution because it removes sessions in a timely manner. We find no long-lived master secrets after applying this patch. Furthermore, the overhead of this solution is fairly minimal, since the second thread spends most of its time asleep, and when it does its work, it’s not correlated in time with any other user events, so the additional system impact is unlikely to be noticeable. Lastly, this patch requires a change only in the AbstractSessionContext class, so the patch is quite minimal to the Android codebase.

All three suggestions solve the master secret retention problem for applications that use standard JSSE Android libraries. However, we find our patched routines are never triggered when the Android Chrome application is running, proving it uses its own TLS routines rather than the system JSSE libraries. However, given that Chrome shows the same retention problem with master secrets, and our scanning tool works on it, it’s likely that Chrome has the same BoringSSL code, but without the same Java layers above. Without the full source code to Android Chrome, we cannot be confident that the fix is as simple as above. A full consideration of Chrome on Android or other desktop platforms is a task for future work.

2.6.2 Observations and Future Work

Conscrypt vs OkHttp

Both Conscrypt and OkHttp are implemented in Java and adopted by Android to support JSSE cryptography. Also, both maintain caches: Conscrypt’s session cache and OkHttp’s connection pool. Conscrypt appears to be the more problematic library, given its lazy deletion of entries in its session cache.

It is interesting to note that the Conscrypt session cache holds sessions which have crucial master secrets, but the OkHttp connection pool holds connections that abstractly
serve one web connection at a time. A future direction for Android might integrate these two libraries together, with a more aggressive and eager deletion process.

**Conscrypt vs BoringSSL**

Conscrypt and BoringSSL together provide TLS implementation on Android. Conscrypt is just a Java wrapper for BoringSSL and is correspondingly many fewer lines of code than BoringSSL, making it much cleaner to read and modify. BoringSSL was intended to be far less “exciting” than its OpenSSL lineage has been with security bugs, but it’s still a large and complicated library with all the security and correctness concerns that apply to any large C codebase.

Given that the problems we found can be considered to be something of a mismatch between Conscrypt and BoringSSL, this raises the question of whether it might be appropriate to change the abstraction boundary, either pushing more work into BoringSSL, or pulling more work out of it into Conscrypt. Certainly, it seems beneficial to store cryptographic keys, themselves, outside of garbage-collected memory, otherwise the GC system could leave behind copies of key material. This raises an interesting development challenge to develop a “minimal” C runtime for key management while doing the rest of the work in Java.

**Coding style**

A common Java coding pattern is to store singleton instances in “static” variables (i.e., global variables). These can never become garbage and will thus never be collected. Any Android app could make such a mistake and accidentally prevent core cryptographic keys from being expired in a timely fashion.

Android Studio, the standard tool for developing Android applications, includes a “lint”
tool with a variety of static analysis inspections that look for common Android coding issues that result in memory leaks. It would make sense to add additional checks that look for incorrect usage of the JSSE cryptographic libraries.

Furthermore, it would be sensible to hide the “real” cryptographic key material behind weak references or some other abstraction that allows the “real” cryptographic keys to be managed without application-layer bugs being able to inhibit the zeroization of expired key material.

A related issue is how to make key zeroization be aware of the Android application lifecycle. We don’t want to create excessive overhead during lifecycle events, but these events do represent significant changes for which it’s appropriate to clean up key material. It might be appropriate to schedule a cleanup activity on a background thread, that can operate at a low priority. Of course, there are times when Android decides that an application must be terminated immediately in order to recover its memory. At that point, it’s too late for an application to clean up its keys. What’s the alternative? Key material could be held in a special memory page, perhaps mapped from a file that Android knows to zeroize and delete as part of application termination.

2.6.3 Android 8

Android 8 was released after this work was complete. Android 8 adds a number of new security enhancements that are relevant to our work, including “background execution limits”. However, we confirmed that the problems discussed in this chapter still exist. Our patches, designed for Android 7 apply cleanly to Android 8 without changes.

The fact that the issues we found apply to many years of Android code suggests that this particular class of attack has not received enough attention. This could also be related to the general limits in static analysis tools with a security focus, which generally only
consider a single programming language, versus the issues here which cross the boundary of C and Java.

### 2.7 Related Work

#### 2.7.1 Memory Forensics

Memory forensics can be largely categorized into acquisition and analysis techniques. Regarding acquisition, Halderman et al. developed “cold-boot attack” [15] showing that an adversary can read out contents of memory, identifying encryption keys from their expanded key schedule; we used a similar technique to identify SSL session structures. Other researchers have looked at vulnerabilities in ARM’s TrustZone [34], allowing a malicious app to obtain full system RAM.

In terms of analysis techniques, signature-based frameworks [35, 36] have been widely used. Various efforts have been made to identify structures by generating robust invariant [37, 38] and using static analysis [39].

A number of authors have looked at Android-specific issues in memory forensics. Sylve et al. [32] first proposed a technique for extracting physical memory from Android devices. Our research utilizes this technique for implementing our test framework. In 2013, Müller et al. [16] showed that cold-boot attacks are also applicable to Android phones.

Memory analysis on Android is commonly focused on extracting sensitive data from applications such as login IDs and passwords. Apostolopoulos et al. [27] showed that login credentials could be recovered from memory images using simple pattern matching. Hilgers et al. [18] identified a variety of data structures in memory images (e.g., GPS coordinates within photo metadata). Thing et al. [40] proposed an automated system that analyzes live memory on Android devices and showed it is possible to extract messages.
DEC0DE [41] proposed a technique to extract plain-text call logs and address book entries from phone storage using probabilistic finite state machines. There have also been studies on specific texting applications such as WhatsApp [29], WeChat [31], and Viber [28]. One clever technique involves recovering previous GUI screens by piecing together the state of the Android widget view hierarchy [42, 43].

To the best of our knowledge, there is no in-depth study of cryptographic secrets of TLS on Android, but, for other platforms, there have recently been several studies for extracting TLS secrets from a virtual machine [20], Windows OS [21], and Oracle’s Java HotSpot JVM [22]. The first two studies look at extracting master secrets; the latter focuses on general data reconstruction from a garbage-collected runtime system.

2.7.2 Android and TLS Security

A full consideration of prior work in Android security is beyond the scope of this work, although Enck et al. [44] provide a nice survey paper. Reaves et al. [45] similarly summarize efforts to apply static and dynamic analysis to Android. Of note, they conclude that no existing tool is suitable for analyzing the cross-language issues that we observed between Java and C. Also, we limit our scope to Android framework and do not attempt to survey the millions of Android applications for how they use TLS. Egele et al. [9] looked at exactly this issue, identifying a large number of Android apps that misuse or misunderstand the correct use of cryptographic APIs.

2.7.3 Mitigation for memory disclosure attack

One of the main causes of data exposure is the insecure deletion which leads to leaking sensitive data [46]. Chow et al. addressed those problems in desktop and server with secure deallocation [47, 48].
One recent trend against memory disclosure attack is to maintain data outside of main memory. Tang et al. [46] suggested CleanOS that encrypts data with a secret key and evicts that key to a secure cloud storage. Also, TRESOR [49] proposed the register-based encryption technique without leaking information into memory. Sentry [50] was developed to maintain data in the cache or internal memory in SoC chip. CaSE [51] was proposed to keep sensitive data using TrustZone from both physical and software-based memory disclosure attacks.

Those studies propose many future directions to mitigate memory disclosure attacks, although maintaining TLS state in a separate piece of hardware will necessarily place a variety of constraints on how it can be used, and as well will create important abstraction boundaries, since a more “distant” store of key state will have less visibility into how those keys are being used.

2.8 Conclusion

In this chapter, we provide an empirical study of Android’s JSEE implementation and its retention of cryptographic secrets. We designed a memory analysis framework that provides physical and logical memory dumping, along with a high degree of automation of experiments. We showed that Android keeps TLS master secret live in memory for an unnecessarily long period of time. Our subsequent in-depth analysis revealed a design flaw in the interaction of Conscrypt and BoringSSL, where Conscrypt maintains a “session cache” that can keep the underlying BoringSSL key material live when it should be zeroized. This issue is further complicated by the interaction of Conscrypt and OkHttp, where the latter maintains a “connection pool” of Conscrypt objects for some time. These issues remain from the oldest Android versions we considered to the latest releases from Google (Android 4 through 8), and they will impact every Android app that uses the standard Android
cryptographic APIs. Luckily, fairly modest patches to these APIs can address the issue for every Android app. We have reported the issues and mitigations described here to Google through their standard security vulnerability reporting process.
Chapter 3

Total Recall: Persistence of Password on Android

3.1 Introduction

In memory disclosure attacks, an unprivileged attacker can steal sensitive data from device memory. These attacks frequently make the headlines: recent vulnerabilities that could lead to such attacks include HeartBleed [23], Meltdown [13], and Spectre [14]. If adversaries can gain physical access to a device, they may also be able to directly dump its memory, e.g., via a “cold boot” attack [15, 16], or even through its USB connection [17]. Memory disclosure attacks pose a serious threat, as sensitive data, such as cryptographic private keys and passwords, is easily reused if stolen (see, e.g., [18]).

Therefore, sensitive data should be deleted from memory as soon as it is no longer in use. Cryptographic libraries have long recognized the importance of this security practice. Some software, such as OpenSSL [52] and GnuTLS [53], zeros out key materials immediately after a session. Of course, garbage collection can complicate matters; the Java Cryptography Architecture (JCA) [54] notably uses char arrays, which can be zeroed, versus String objects, which are immutable.

Of course, sensitive data exists beyond cryptographic key material, and applications that handle secret data also go beyond cryptographic libraries. In this study, we particularly focus on one type of sensitive data—user passwords—and how they are used in practice by real Android apps. Although other authentication mechanisms have been proposed (see, e.g., [55]), password-based authentication is still the de facto practice for many applica-
tions. Some apps transmit passwords across the Internet in plaintext, with or without SSL, to servers that compare those passwords against plaintext databases. Others may derive cookies or authentication tokens from passwords. Yet others may derive master keys from passwords using “key stretching” [56] techniques including PBKDF2 [57], bcrypt [58], or scrypt [59], ensuring that a network adversary cannot learn a reusable plaintext password.

Cryptographic libraries have integrated many well-understood security practices (e.g., constant-time cryptography [60]), and developers tend to stick to relatively mature libraries (e.g., OpenSSL). However, when it comes to password-based authentication, a developer may be tempted to roll her own version with idiosyncratic security practices. In these apps, it is up to individual developers to properly manage and delete passwords. Naturally, we would expect many apps to have poor security practices. In fact, existing apps have been reported to misuse the TLS library [6, 7, 8], cryptographic APIs [9], OAuth protocols [10], and fingerprint APIs [11]. A recent study has also revealed that many developers simply store passwords in plaintext on disk [61]. In this chapter, we ask an important question: *How well do Android apps manage passwords in memory?*

The Android platform has many complex layers that interact, such as the Dalvik/ART runtime system, including its garbage collector, which coexist with the Android application lifecycle, where an app might be put into “background” or even “stopped” while remaining an active process holding allocated memory. In these circumstances, even an app intending to clean up sensitive values in its memory might end up with those values having an undesirably long lifetime. Additionally, user passwords go through a long chain of custody before authentication takes place, sometimes even passing from one app to another via IPC / Binder calls. Each of the steps in this chain may inadvertently retain passwords longer than necessary. Last but not least, previous studies have found that Android apps fall short in performing secure deallocation [46], and that they may retain TLS key materials in
Using system memory dumping and code analysis, we have found that many popular apps, including a banking app, password managers, and even the Android lockscreen process, retain user passwords in memory long after they are not needed. The passwords can be easily extracted from memory dumps using simple scripts. This is a serious problem, because users often use similar or identical passwords across applications [63, 64, 65], so a stolen password would cause widespread damage. We have also identified common root causes, such as Android’s password widget, which doesn’t erase its internal memory, and the widespread use of String objects to store passwords. We propose solutions that fix the Android framework and the studied apps with modest code changes: SecureWidget and KeyExporter. SecureWidget distinguishes password inputs from regular texts, and carefully manages the password buffers to avoid retention in the Android framework. KeyExporter, on the other hand, can be integrated with individual apps to eliminate password retention. It achieves this by integrating cryptographic primitives directly within the password widget, using password-based key derivation functions (e.g., PBKDF2, scrypt) and password-authenticated key agreement (e.g., SRP [66]). Our evaluation shows that these solutions eliminate password retention in all of the apps that we tested, hardening the system against memory disclosure attacks, while placing only a modest burden on the developer to adopt better practices.

Concretely, after describing more background in Section 3.2, we make the following contributions in this chapter:

- A demonstration of the password retention problem by analyzing memory dumps of 11 popular apps (Section 3.3);
- A comprehensive analysis of the Android framework and a variety of Android apps to identify common root causes (Sections 3.4+3.5);
• Our solutions: SecureWidget and KeyExporter, helping Android app authors eliminate password retention and follow stronger cryptographic practices (Section 3.6);

• Implementation and evaluation of our solutions, which successfully achieve our goal of timely password deletion in all tested apps (Section 3.7);
3.2 Background and Motivation

In this section, we describe more background on Android authentication, as well as how passwords may be retained for longer than necessary at each stage of an Android app lifecycle. Then, we will discuss the key questions we would like to answer in this study, as well as the technical challenges.

3.2.1 Authentication in Android

Recent versions of Android have added the use of fingerprints, face recognition, and voice recognition as means of authenticating an Android user to their devices, including APIs that can connect these biometrics to cryptographic hardware within the device for remote attestation. However, to date, passwords are still the mainstay for Android app authentication, so we have focused on them as our main target. Broadly, Android user authentication falls into two categories: remote authentication, where an app needs to send some secret to a remote server (e.g., social networking apps), and local authentication, where authentication is handled entirely on the local device (e.g., password managers or the lockscreen app).

Remote authentication. Figure 3.1 shows the workflow of remote authentication, which has three main stages. ① The app prompts the user to enter her password, and then contacts the remote server with the user credential. The server validates the credential and returns a cookie or authentication token upon success. ② The app receives the cookie or token, which will be stored in a secure location (e.g., private files of the app) and used for further requests to the server. ③ Whenever the app needs to contact the server again, it looks up the token from the secure storage, and resumes the session without prompting for the user password again. The user will not need to enter her password again until the shared
Figure 3.1: Authentication steps in client applications.
Notice that, in the workflow above, only the first stage involves user passwords. Such a design helps security, as it minimizes the exposure of passwords. This also means that there is no need for an app to retain passwords once it reaches the end of the first stage. However, in practice, the first stage also tends to be quite complicated. Some apps may use a challenge/response authentication. Others may use a password-authenticated key exchange (PAKE) protocol to generate a zero-knowledge proof of knowledge of the password. Or, if an app uses OAuth, this would further involve a relying party, which wants to verify a user’s identity, as well as an identity provider party, who has a record of the user identity (e.g., Facebook or Google). In order to achieve a secure design, the app needs to ensure that user passwords are deleted properly, despite all the complexity in this stage.

**Local authentication.** Sometimes, apps only require local authentication without involving a remote server (e.g., the lockscreen app, or password managers). Such apps obtain user passwords and use them to encrypt or decrypt local data as well as to authenticate users. Password managers, for example, store sensitive information, such as bank account passwords and passport numbers, in a local encrypted database. Users interact with the password manager via a “master” password, and the app then derives a strong cryptographic master key, e.g., using key stretching [56]. Such apps can also help users generate random passwords to mitigate password reuse [63]. As concrete examples, two popular apps in this category are 1Password* and LastPass†, which use PBKDF2-HMAC-SHA256 for key derivation. Needless to say, the security of these applications hinges critically on the protection of the master passwords. If they are retained in memory after use, the entire password database would be vulnerable to compromise.

---

*See [https://support.1password.com/pbkdf2/](https://support.1password.com/pbkdf2/)

†See [https://lastpass.com/support.php?cmd=showfaq&id=6926](https://lastpass.com/support.php?cmd=showfaq&id=6926)
3.2.2 Opportunities for Password Retention

Unfortunately, there are many opportunities for passwords to be retained by Android for longer than necessary. We briefly discuss some of them below.

**Background applications.** The activity lifecycle of Android apps is different from that of traditional desktop programs. Android apps can be “paused” and then “stopped” when they are switched to the background, and they can be “resumed” and “restarted” when switched back. When an app goes to the background, its GUI is no longer visible to the user; but it remains an active process in the system. When it is resumed, its GUI will be displayed again to the user. Therefore, if an app is still holding user passwords when it goes to the background, the passwords may live in memory for an extended period of time. Although Android may destroy certain background processes, it typically does so only if the system resources are running low.

**Lack of special support for password protection.** Over the years, Android has been integrated with many security features, such as the ARM TrustZone platform. A program that runs in the “secure world” inside TrustZone will be protected from attackers in the normal world, so secret data will not be visible to external programs. Android uses this feature for many security applications, such as its Keystore service and fingerprint authentication. Key materials and fingerprint data are stored inside TrustZone. As such, the data is protected from memory disclosure attacks that exploit software vulnerabilities, or even from attackers with root privilege. However, Android does not offer the same level of separation for user passwords.

**GC and Strings.** Most Android apps are written in Java, so their memory is managed by a garbage collector (GC), meaning that even when an app deletes its last reference to a password, that password will still exist in memory until the GC system reuses the memory. This delay can span minutes or hours, depending on the allocation pressure on the GC
system. Furthermore, in the (seemingly intuitive) case of using Java’s class `String` to hold passwords, developers cannot manually overwrite them, because `String` objects are immutable. The ubiquitous use of Java strings in libraries of all kinds (e.g., JSON data import and export), means that even if an app author wishes to use `char` arrays, as JCA uses internally for cryptographic purposes [54], they may find it difficult to make the switch. (We show consequences of this in Section 3.5.)

**Java vs. native (C) code.** Although Android apps are commonly written in Java, they may make native calls to any underlying C library, either included with the app or installed natively on the system. For example, the Android TLS implementation wraps a Java layer (Conscrypt) atop the BoringSSL cryptographic library (written in C). If passwords are copied from the Java layer to the C layer, there is also a possibility for the data to be retained in the C layer [62].

**Long chain of custody.** User inputs may also be unintentionally buffered by various processes and retained in memory. For instance, when a user inputs her password, the keystrokes traverse multiple processes: first the input device driver, then the keyboard app, and finally the user application that prompted for the password. By the time the password reaches the intended app, it has been touched and possibly copied by multiple processes, any of which might have accidentally retained the password, perhaps by virtue of GC issues, or perhaps for other reasons.

### 3.2.3 Key Questions

The rest of this chapter is centered around two inter-related research questions: (i) *How well does Android handle user passwords?* In order to perform analysis at scale, we adopted a blackbox approach: for each app, we simply dump its memory and use simple scripts to search for retained passwords. We have found all of the tested apps to retain passwords in
memory, even the most security critical apps, such as a banking app and password managers.

The second research question is (ii) *How can we develop effective remedies to harden Android against memory disclosure attacks?* To this end, we perform a comprehensive analysis of the Android platform and various sample apps using a whitebox approach, where we manually inspect the source code (when available) of the apps and the Android platform to identify common root causes. We then develop fixes to address each of the root causes, and validate that the proposed fixes effectively eliminate passwords.
3.3 Black-box Analysis of Memory Dumps

In this section, we describe our methodology for an initial black-box analysis. Our analysis is based on taking a memory dump of an Android app after the authentication step, and then using customized scripts to extract passwords from the dump. We also present our findings on 11 apps, finding password retention for all of them, including apps with more than one billion installations (e.g., Gmail, Chrome), and apps that directly process highly confidential data (e.g., the Chase banking app).

3.3.1 Methodology

Our threat model and analysis methodology are as follows.

**Threat model:** We assume that an adversary can perform memory disclosure attacks on an Android device, e.g., through exploiting software vulnerabilities, launching side-channel attacks, or after the physical capture of a device. This threat model is similar to that in previous studies [18, 27, 46, 62]. To set up such a scenario and facilitate our analysis, we depend on two tools used by previous studies: (i) a native program, pmdump [62], which can extract data from the address space of a process; using this tool, we can precisely identify the location of data in the process memory (e.g., in the JVM heap, the native heap, or the native stack); and (ii) a kernel module, LiME [32], which can dump the full physical memory image including kernel memory.

This threat model assumes a very powerful attacker, but as we discussed earlier, modern cryptographic libraries are regularly engineered to mitigate against such attackers, and it’s reasonable to expect a similar level of engineering effort to protect passwords against the same threat.

**Our setup:** In order to achieve a thorough understanding across Android versions, we used three different environments: two different versions of emulators running Android 7 and 8,
and a Nexus 5 device running Android 6. We tested 10 Android apps on each environment. Six of the apps are very popular—having more than 10 million installations each—and the other four apps are password managers that store highly sensitive user data. Additionally, we tested the system processes—SystemUI and LocalSettingsService—that are in charge of unlocking the phone after receiving the correct password.

The workflow of our analysis: We installed and launched each app, and manually entered passwords for authentication. After this, we performed a full physical memory dump as well as a per-process dump. If an app has deleted passwords promptly, we would see no password traces in the memory dumps; if an app has retained passwords, they would show up in the memory dump. We looked for password encodings in two-byte characters (UTF16, as used by the Java String object), as well as one-byte characters (as used by ASCII). We performed such a dump several times for each app: a) right after authentication (“login”), b) after moving the app to the background (“BG”), c) after additionally playing videos from the YouTube application (“Youtube”), and d) after locking the phone (“lock”).

3.3.2 Results: Memory Dump Analysis

Table 3.1 shows our results obtained by analyzing the memory dumps. We note four high-level observations.

Observation #1: All tested apps are vulnerable. As we can see, none of the apps that we tested does a good job in cleaning up passwords. Even very popular apps, such as Facebook, Chrome, and Gmail, which have been installed more than one billion times, retain passwords in memory. In the case of password managers, the user passwords are typically used to decrypt their internal password databases, so an attacker would be able to capture the password and gain access to the full databases. Similarly, the lockscreen process also leaves the PIN passwords in memory. Since the PIN password is used for full-disk
<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Installs</th>
<th>Login</th>
<th>BG</th>
<th>Youtube</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gmail</td>
<td>email</td>
<td>1,000 M</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Chrome</td>
<td>browser</td>
<td>1,000 M</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Chase Bank</td>
<td>finance</td>
<td>10 M</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Facebook</td>
<td>social</td>
<td>1,000 M</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Tumblr</td>
<td>social</td>
<td>100 M</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Yelp</td>
<td>social</td>
<td>10 M</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1Password</td>
<td>password</td>
<td>1 M</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dashlane</td>
<td>password</td>
<td>1 M</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Keepass2Android</td>
<td>password</td>
<td>1 M</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>passwdSafe</td>
<td>password</td>
<td>0.1 M</td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Unlocking phone</td>
<td>system</td>
<td>N.A.</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.1: Exposure of passwords in application memory.

encryption and decryption, Android spends an extraordinary amount of effort in protecting the PIN—e.g., the GateKeeper service verifies hashes of user passwords in TrustZone to protect the original passwords. Therefore, in the presence of memory disclosure attacks, the retention of PIN passwords in memory completely defeats the purpose of the added security measures. Interestingly, we observe that some apps seem to have already paid attention to address password retention: Keepass2Android and Dashlane only leave one or two passwords in memory, and the lockscreen process removes passwords after about an hour\(^\d\). Nevertheless, the general picture we see from this table is worrying evidence that

\(^\d\)A previous study considers a piece of data to be exposed if it persists in memory more than ten minutes [46].
Android apps, as well as the framework itself, are vulnerable to memory disclosure attacks. **Observation #2: All tested Android versions are vulnerable.** Although Table 3.1 only presents the results from Android 8, we have found that Android 6 and 7 both retain passwords, and the only difference is the number of password copies. This implies that password retention is not specific to a particular Android version. For the rest of this chapter, we only present our findings on Android 8.

**Observation #3: Password strings are easily recognizable.** For many applications, we have found password strings together with other easily recognizable string patterns. For instance, the Facebook app contains ASCII string patterns like ...&pass=1839172..., and the Tumblr app has p.a.s.s.w.o.r.d.=.1.8.3.9.7.2. in UTF16 encoding. In addition to the full password matches presented in Table 3.1, we have also found fragmented passwords. For instance, a full password mypasswd may be fragmented as mypassw or mypass. This is because of the use of SpannableStringBuilder, which we will describe in more detail in Section 3.4.

These partial strings support our hypothesis that Java programmers tend to use strings to construct messages before sending them to the network. The Facebook fragment suggests that the Facebook app is constructing a URL for talking to its web service endpoint and is sending the password to their service in plaintext, albeit within a TLS-encrypted session.

**Observation #4: Some developers have paid attention to the password retention problem.** We can see evidence that some apps (e.g., Chase Bank, Dashlane, and Keepass2Android) seem to be actively clearing out the passwords. For these three apps, the passwords disappear once we have put them into the background. Although the same apps are still vulnerable before being put into background (which still represents a window of opportunity for attacks), this suggests that a) the problem of password retention seems to have gained attention from at least some Android developers, b) but the approaches need to be more
thorough in cleaning the passwords, and c) ideally, we should develop fixes that are general enough to work across different applications.

Overall, the above findings are worrying evidence that password retention is a widespread problem in Android. Notably, Tang et al. looked at similar issues on Android 2.3.4 in 2012 [46], finding four out of 14 apps left passwords in memory. The problem seems to have worsened today; Overlapping with our work, Tang et al. also looked at Facebook and Gmail, concluding that they had no problems, but both apps have problems today.
3.4 In-Depth Analysis: Android Framework

In the next two sections, we perform an in-depth analysis of the password retention problem and identify its root causes. Our goal is to distill common security practices for password management, both good and bad. For clarity of presentation, this section focuses on the Android framework, and the applications that typically come with the framework (e.g., keyboard apps); the next section focuses on Android applications.

3.4.1 Methodology

In order to achieve a thorough understanding of the root causes, we have manually analyzed the Android source code, made changes to the codebase, and tested the effect of a change. We have used two key techniques: runtime logging, and password mutation.

Runtime logging: We annotate core modules in Android, using the standard logging facility, giving us a timeline of the use of function calls related to password processing.

Password mutation: In order to precisely pinpoint the location of password retention, we also apply password mutations as the passwords pass through different Android components. When a component $C_i$ receives a password $p_i$, it will index a pre-defined permutation dictionary using $p_i$, and obtain a mutated password $p_{i+1}$ before passing it to the next component $C_{i+1}$. Therefore, when we take the memory dump, we know that instances of $p_i$ are hoarded by component $C_i$, whereas instances of $p_{i+1}$ are hoarded by $C_{i+1}$. Our algorithm also ensures that these password mutants always have the same length, so we can easily locate a password fragment within the component that’s using it. While this approach is quite simplistic, it’s more than adequate for tracing out how Android’s various subsystems handle passwords.

We ruled out an automated approach, e.g., using static analysis [67, 68, 69] or dynamic analysis tools [70, 71], for two reasons. Fundamentally, a static analysis tool can only
find a code pattern that we know in advance, but our study is in nature exploratory. We need first to understand the root causes before we could automate detection and analysis. Second, typical static analysis tools focus on a single language like Java (see, e.g., [45]), but we need to track passwords beyond the Java layer into the native stack/heap as well as the kernel itself.

3.4.2 The Android Framework

Figure 3.2 shows the flow of data in Android when a user types her password. The signals from the touchscreen are transmitted to a software keyboard app, otherwise known as an input method editor (IME) app, via the kernel driver. Then, the keyboard/IME app will send the password to the UI widget (e.g., TextView) in the application (e.g., Facebook) via a dedicated input channel. The UI widget also stores the password internally, so that it could pass the data to the application upon request. Additionally, the widget sends the data to a graphics module, so that the input strokes are echoed back and displayed on the screen as stars (*) by the display device driver. Any unintended buffering or mistake in any of the stages would cause password retention. Interestingly, only ⑤ is managed by developers, whereas all other stages are built into the Android framework.

After testing all these stages, we were able to narrow down the culprit to the UI widget and keyboard apps, because all the password mutants and fragments we captured corresponded to the versions between ③ and ④. Subsequently, we further analyzed the source code of the UI widget, and found that Android does not implement a dedicated class for password widgets, but rather simply reuse the TextView class. This class contains about 12,000 lines of code (LoC); as it is not designed exclusively with passwords in mind, its codebase contains many instances of insecure password handling.

There is a flag in the TextView class that indicates whether it is a regular text field or a
password field, but this flag only affects whether a character is echoed back as a * or not, and whether the text can be copied or selected by a user. All other management of the input uses the same logic for regular text and sensitive passwords. Since passwords aren’t given any particularly special handling, we shouldn’t be surprised if there are problems. Indeed, we have identified three problems with the `TextView` class.

**Problem #1: The lack of secure zeroization.** First, the `TextView` class does not zeroize or otherwise erase the buffer when an app is “paused”, “stopped”, or even “destroyed”. Therefore, when one of these lifecycle activities takes place, the memory object that holds the text remains intact. This puts the responsibility for secure zeroization solely with the app developers, who would need to deal both with the application lifecycle as well as with zeroizing the `TextView` buffer after login completes. We argue that this responsibility should be handled within the `TextView` rather than by the app developer.

**Problem #2: Insecure `SpannableStringBuilder`.** The buffer in a `TextView` class is ac-
tually a `SpannableStringBuilder`, whose implementation leads to two problems. First, whenever a user types a new character of her password, `SpannableStringBuilder` will allocate a new array, copy the previous password prefix to this array, and discard away the previous array without clearing it. This is actually the root cause of why we see fragmented passwords in memory. We note that the `SpannableStringBuilder` class provides a clear method, but it simply sets the internal data to `null` rather than zeroizing the data. If an app developer mistakes this method to mean “secure deletion”, the password may still remain in memory.

**Problem #3: `toString()` considered harmful.** Developers typically obtain the contents of a `TextView` object by invoking its `getText()` method, which returns a `CharSequence` interface instead of a `SpannableStringBuilder` object. Since `CharSequence` is also the interface of the `String` class, developers often treat it as a kind of `String`, and invoke the `getText().toString()` method to turn the password into a `String` object. Strings, however, are known to cause security problems: The official JCA library [54] specifically suggests that “Objects of type `String` are immutable, so there is no way to overwrite the contents of a `String` after usage”; it further suggests that developers should not store passwords in `String` objects.

On a related note, in the Java Swing UI library, the equivalent password widget is called `JPasswordField`, which also has a `getText()` method that returns a password in a `String` object. However, this feature was deprecated as part of Java 1.2 in 1998, with the suggestion replacement of `getPassword()`, which returns a character array. Unfortunately, the Android version lacks such a `getPassword` API. When an Android developer uses the `getText().toString()` method to obtain passwords, the resulting strings will then be harder to erase.
<table>
<thead>
<tr>
<th>Application</th>
<th>Company</th>
<th>Installs</th>
<th>ID</th>
<th>PW</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LatinIME</td>
<td>Google</td>
<td>N.A.</td>
<td>0</td>
<td>2</td>
<td>Default in AOSP</td>
</tr>
<tr>
<td>Gboard</td>
<td>Google</td>
<td>500M</td>
<td>1</td>
<td>0</td>
<td>Default in Android 8</td>
</tr>
<tr>
<td>Swift</td>
<td>SwiftKey</td>
<td>300M</td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Go</td>
<td>GOMO Apps</td>
<td>100M</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Kika</td>
<td>Kika AI Team</td>
<td>100M</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TouchPal</td>
<td>TouchPal</td>
<td>100M</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Cheetah</td>
<td>Cheetah</td>
<td>50M</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>FaceMoji</td>
<td>Facemoji</td>
<td>10M</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>New Keyboard</td>
<td>2018 Keyboard</td>
<td>10M</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Simplified Chinese</td>
<td>Linpus</td>
<td>0.1M</td>
<td>117</td>
<td>135</td>
<td>Chinese</td>
</tr>
<tr>
<td>Simeji</td>
<td>Baidu</td>
<td>10M</td>
<td>0</td>
<td>0</td>
<td>Japanese</td>
</tr>
<tr>
<td>TS Korean</td>
<td>Team Space</td>
<td>0.01M</td>
<td>3</td>
<td>0</td>
<td>Korean</td>
</tr>
<tr>
<td>Baidu Voice</td>
<td>Baidu</td>
<td>0.1M</td>
<td>0</td>
<td>2</td>
<td>Voice support</td>
</tr>
</tbody>
</table>

Table 3.2: Results for the tested keyboard applications.
3.4.3 Keyboard (IME) Applications

Next, we proceeded to analyze the input channel between TextView and keyboard apps. Although we found that the input channel is tightly coupled with the buffer of TextView, the channel itself, fortunately, does not perform additional buffering.

But what about the keyboard app? Android has a default keyboard, and it also provides extensions that allow any developers to build their own keyboards. This feature is very useful, and has led to a rich ecosystem of third-party Android keyboard apps, variously innovating in how they predict words, how they handle gestures, and how they handle input of accent characters, non-Latin alphabets, and emoji. Of course, a keyboard app is also central to the entry of passwords, so any interaction of the keyboard’s internal features, like saving prior words for future predictions, must be careful not to save password entries.

We selected popular keyboards apps, as well as ones that support special features, such as voice inputs or different languages. Also, we tested the LatinIME keyboard, which is the default keyboard in Android Open Source Project (AOSP). All tested apps are listed in Table 3.2. We examined many popular keyboard apps, with hundreds of millions of installs (Gboard, SwiftKey, Go, Kika, TouchPal) as well as a number of less popular keyboards.

For each keyboard, we used it with the Facebook app, typing our Facebook credentials for login. We then moved Facebook into the background and locked the phone before performing a memory dump. Table 3.2 shows the number of copies of user account (ID) and password (PW) that we discovered for each keyboard app. Out of the 13 apps we tested, nine of them hoarded user passwords. Fortunately, two of the most popular keyboards, Gboard and SwiftKey, cleaned up the passwords perfectly, only buffering account IDs in memory which, generally speaking, aren’t treated like passwords so their presence in memory isn’t security-relevant. Our most worrisome example, on the other hand, is a keyboard that supports simplified Chinese—it kept more than 100 copies of the user password in
memory.

**Problem #4: Buffering the more recent input.** Surprisingly, the LatinIME keyboard also has retention, so we further analyzed its source code to identify the root cause. We found that this is because this keyboard buffers the most recent input obtained from a user, and only clears this buffer upon the next user input. For instance, if a user unlocks the phone with her PIN code, or if she enters the password to a banking app, her password will be retained by this keyboard until the next time she types in something else. This is a serious problem, because if the last activity of a user is to enter any sensitive information via the keyboard, such information would be susceptible to memory disclosure attacks.

The insecurity with the LatinIME keyboard, in particular, is problematic: many third-party keyboard apps may model themselves after this default AOSP keyboard. This password buffering coding pattern, may, therefore, be inherited from the LatinIME and create problems in many derivative keyboards. Although it is difficult to trace the provenance of the closed-source keyboard apps, we believe that Table 3.2 hints at this practice. The Go, Kika, FaceMoji, New Keyboard, and Baidu keyboards all have similar patterns as the LatinIME keyboard: they hold the user passwords but not the account IDs. This is because users typically first type their account IDs and then passwords, in that order. Since the earlier entry is gone and the later entry is present, this suggests reuse of LatinIME in some fashion.
3.5 In-Depth Analysis: Android Apps

Next, we analyze several Android apps, where individual app developers read in user’s passwords and, thus, should be responsible for erasing them. We looked at four categories of apps: a) basic password-based authentication apps, which simply send user passwords to a remote server for authentication, b) challenge/response apps, which derive secrets from passwords for authentication, c) OAuth apps, which delegate authentication to an OAuth service (e.g., Facebook), and d) local authentication or standalone apps that do not involve a remote server, including some password manager apps.

3.5.1 Methodology

One challenge that we face in analyzing Android apps is that many popular apps are closed source. For apps without source code available, our approach is to obtain similar types of apps from official sites (e.g., Google’s Play Store), open source repositories (e.g., GitHub), security guidebooks, and developers’ website such as Stack Overflow. For open-source Android apps, such as Keepass2Android and passwdSafe, we directly analyze their source code. Collectively, we have analyzed more than 20 apps. Many of the apps we examined are relatively simple, demonstration apps for some particular functionality rather than full-featured applications. However, existing studies suggest that an analysis of these sample apps is more representative than one might initially think: many real-world apps contain the same snippets from sample apps, e.g., obtained from Stack Overflow [72]. Table 3.3 summarizes our results.

3.5.2 Basic Password-based Authentication Apps

The first type of apps we have analyzed is basic authentication apps, where an app sends raw user passwords to a server via HTTP/HTTPS. The majority of apps we collected fall
<table>
<thead>
<tr>
<th>Category</th>
<th>Application</th>
<th>Description</th>
<th>Login</th>
<th>BG</th>
<th>YouTube</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Sample 1</td>
<td>Uses Volley lib.</td>
<td>25</td>
<td>24</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Basic</td>
<td>Sample 2</td>
<td>Uses Apache lib.</td>
<td>28</td>
<td>16</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>Basic</td>
<td>Sample 3</td>
<td>Empty request</td>
<td>11</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Basic</td>
<td>Sample 4</td>
<td>Nullifying widget</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cha./Resp.</td>
<td>Sample 5</td>
<td>App security book</td>
<td>21</td>
<td>20</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>OAuth</td>
<td>Sample 6</td>
<td>With Facebook</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OAuth</td>
<td>Yelp</td>
<td>With Facebook</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.3: Results for the tested sample apps.

into this category, because directly sending passwords is the simplest (but an insecure) way of authentication. These apps use different libraries for their implementation, but they share similar authentication steps. Sample apps 1-2 in Table 3.3 are some of the basic password-based authentication apps we have tested, and they use different libraries for network communication (Apache vs. Volley). The results show that both of them have many password copies. Compared to the apps we have examined in Section 4.2, these apps are far worse. In order to understand why such simple apps have so many password copies, we have modified the apps one by one, and analyzed how our modifications make a difference.

**Problem #5: The use and propagation of string passwords.** We started by focusing our attention on their use of string passwords. As we mentioned in Problem #3, some apps obtain a String type of user passwords from TextView using getText().toString(). To measure the effect of String usage, we deleted all uses of String in sample app 1, and instead, we made the app to send an empty password. The “server”, on the other hand, was
based on a local desktop machine in our lab; it always sent successful authentication messages to the app. Sample app 3 in the same table shows the results after the modification: as we can see, the number of password copies decreased by more than half compared to the original version (sample app 1). This demonstrates that String is a major cause, though not the only one.

**Problem #6: Lack of manual TextView cleanup.** This is the equivalent of Problem #1 manifesting itself in the Android apps. Recall that the clear() method in the buffer of TextView does not actually implement secure deletion, but rather sets the buffer to null and leaves it to the garbage collector. Nevertheless, we modified the sample app 1 to call this method right after the login to see how effective this is. Sample app 4, which is the modified app, did show some more improvement: all passwords in the app’s memory were successfully deleted after playing the YouTube videos or locking the phone, though they still resided in memory after login and after the app went to the background. Therefore, calling the clear() method of the buffer is indeed effective to some extent. An app that calls clear() method in the buffer of TextView to remove passwords would be more secure than an app that does not.

**Problem #7: The lack of app-level zeroization.** Even if an app developer uses char arrays instead of String objects, she would also need to clean up the passwords in their apps manually, e.g., by zeroing out the char arrays. In other words, even for a perfectly secure TextView implementation, developers may still accidentally hold passwords in memory if they do not deliberately clean up password memory.

### 3.5.3 Challenge Response Authentication Apps

We now turn to analyze authentication apps that are based on a challenge/response mechanism. Challenge/response apps do not directly send passwords to the network, but rather
generate HMAC values from user passwords and use them as secrets for the remote servers to perform authentication. In this regard, they are more secure than sample apps 1 and 2. So, would such apps handle passwords in a more secure manner? We took an app from a popular security guidebook [73] and performed our test on this app (sample app 5). As shown in Table 3.3, this app is not notably different from sample apps 1 and 2. In fact, we analyzed the source code and found the same problems #5-#7 in its source code. Even worse, this “security guidebook” app simply uses strings for passwords. Although the number of password copies is slightly lower than these in apps 1-2, this reduction of password copies mainly comes from the fact that passwords are not propagated as widely as in the apps that directly use passwords for authentication.

3.5.4 OAuth Authentication Apps

We next consider apps that provide OAuth services. Since Facebook is a dominant identity provider for OAuth, we have implemented sample app 6 using the Facebook OAuth library by following the official guide from Facebook. If we launch the app and click the login button, our app redirects to the Facebook app, which then prompts for password inputs and performs authentication; the Facebook app then redirects to our original app upon success, which displays a successful login message.

Sample app 6 in Table 3.3 shows the results. Although it is far more secure than previous apps, it still holds quite a few password copies in memory, and one of the copies remains in memory long after the phone is locked. Moreover, all passwords were found only in the memory of the Facebook app itself, but none in our sample app.

Since the Facebook OAuth library is not open source, we were not able to perform a code analysis. However, our instrumentation of the Android framework reveals that Facebook uses the standard TextView for passwords, as well as its toString() method, which
explains the password retention. This is unfortunate, because Facebook’s OAuth library is completely outside of the reach of any developer who might want to zeroize passwords in its memory, and this issue will impact any app that uses it. To demonstrate this, we tested the Yelp app, which uses Facebook OAuth, and Table 3.3 shows identical password retention for Yelp as for sample app 6.

3.5.5 Password Managers

We now consider password managers. As shown in Table 3.1, password managers are comparatively more secure than other apps, yet still many passwords remain in memory. To begin, we analyzed the code for Keepass2Android and passwdSafe, which are two popular open-source password managers, to identify their practices for password handling. Keepass2Android consists of more than 80,000 lines of code and more than 300 source code files. We found that its codebase is particularly well engineered to handle secure password deletion.

- It converts a password into char array and uses the latter to generate master keys.
- After authentication, it sets the TextView’s content to an empty string, and it manually sets all password-related objects to null.
- It manually invokes the garbage collector, which might help accelerate memory reuse.

Using a combination of the above techniques, Keepass2Android successfully cleared all passwords from its own memory when it went to the background. However, Keepass2Android still obtains passwords from TextView as a String object before the char conversion, which results in immutable passwords outside of the app’s ability to erase them. Indeed, we found passwords for Keepass2Android in the full memory dump.
Another password manager, passwdSafe, is also commendable in its security measures. In fact, we were impressed with their level of effort in handling passwords after analyzing its source code. First, this app implements a manual reference count for passwords, instead of depending on automatic memory management in Java itself. When passwords are copied, the reference count will increment. When copies are released, the reference count decrements. When a reference count drops to zero, its cleanup method *overwrites the password three times with different values*, which seems unnecessarily paranoid. Unlike Keepass2Android, passwdSafe does not seem to be aware of the TextView problem, and there is no attempt to clear its buffer.

The results for password managers are clearly more encouraging than other apps, as we can see considerable efforts gone into secure management of passwords. However, despite developers’ best efforts, we can see that even experts with intricate knowledge of security practices fail to completely solve the password retention problem. This is because there are many opportunities for retention; as a result, overlooking even one of them would cause password to be left in memory.

### 3.5.6 System Processes

Last but not least, we turn to a special type of authentication in the Android framework: PIN authentication in the lockscreen service. As shown in Figure 3.3, this system service in Android is designed with security in mind, as it leverages several services in the secure world in TrustZone, such as Gatekeeper and Keymaster. Unfortunately, we found that the SystemUI and LockSettingService processes, which are in charge of PIN authentication, also have password retention. The SystemUI process uses the standard TextView to obtain passwords, and like other apps, it also converts the password into a String object. This process then sends the String object directly to the LocalSettingsService process via
Figure 3.3: Authentication steps when unlocking a device.

Binder. The LocalSettingsService finally converts the string password into a char array, and derives keys from this password using scrypt; these derived keys are further protected by TrustZone. However, the original password, as it is stored in a String object, is immutable and survives in memory long past its use.

This is an unfortunate but classic example where a single weak link can break the entire security chain. Even though the developers for the Android framework explicitly keep security in mind, using security features not available to regular Android developers, the TextView class and the use of String-based passwords leaves a prominent vulnerability.

3.5.7 Summary of the Problems

To summarize, our analysis has revealed seven main causes of password retention, which are prevalent in many apps and even the Android framework itself. Some of the root causes
are also inter-related. For instance, Problem #5 (the use of String passwords) need to be addressed before Problem #7 (manual zeroization) can be solved, because String objects are immutable. As another example, Problem #5 can be attributed partly to Problem #3 (the insecure design of TextView). Therefore, in order to arrive an effective solution, we need to develop a systematic fix that addresses all of the identified root causes together.
3.6 Our Solution

Our analysis suggests that password management is very challenging to get right. Tackling this challenge should not be something that is left to each individual developer. In this section, we describe our solution to these problems in the design of SecureWidget and KeyExporter, which can successfully remove unneeded passwords from Android’s memory.

3.6.1 SecureWidget: Fixing the Android Framework

We have developed fixes for the Android framework by designing a secure version of TextView that we call SecureWidget. Since the root cause of retention in the Android framework lies in the use of TextView, we developed fixes to TextView to handle passwords differently from regular textual inputs, by zeroing out sensitive memory after use. This addresses Problem #1. SecureWidget also implements a secure version of SpannableStringBuilder, which is the buffer type used in TextView, to avoid password fragments from being left in memory. This addresses Problem #2. We defer the fixes for Problem #3 to a later section in the design of KeyExporter, which will address the use of String objects altogether. This is because the use of String objects is also prevalent in the Android apps, so a deeper treatment is necessary.

SecureWidget is different from the regular TextView in that it treats password inputs differently from regular inputs, and also that it fixes the insecure design of the SpannableStringBuilder class. When SecureWidget processes a password field, it uses a secure implementation SecureBuffer instead of SpannableStringBuilder as the buffer. The design and implementation of SecureBuffer closely follow these of SpannableStringBuilder, but it avoids leaving password fragments and it contains a secure close method that cleans up passwords. Moreover, SecureWidget has an event handler that listens for status changes of the phone. If the phone is locked, or if an app becomes inactive, SecureWidget au-
tomatically zeroizes its buffer to clean up password fields. SecureWidget can be used as a drop-in replacement of TextView, as all code changes are localized to the implementation of TextView and SpannableStringBuilder. Overall, SecureWidget differs from the regular TextView only by 500 lines of code in Java.

To fix the buffering problem in keyboards (Problem #4), we modified the code of the open-source keyboard app LatinIME to avoid holding on to the most recent user input. Since the other keyboard apps are not open source, we could not easily modify the source code and test the fix. Nevertheless, as we discussed before, we observed similar behaviors in these apps as LatinIME, so they are likely caused by a similar buffering problem. At the time of writing, we plan to report this vulnerability to each vendor individually, with our suggestions and the LatinIME patch we have developed. Since LatinIME serves as example for many third-party keyboard apps, we believe that this patch would be valuable to a wider audience of keyboard app developers. In addition, we plan to contact Google to update its official documents that describe security issues with creating keyboards; the current documents only suggest that passwords should not be stored in a file, but they should ideally also include suggestions on avoiding password inputs from being buffered.
3.6.2 KeyExporter: Fixing Android Apps

Problems #5-#7 (as well as #3) need to be addressed on a per-app basis, since it is up to an individual app developer to avoid insecure engineering practices. These practices include: using char array to hold passwords obtained from the password widget instead of String, clearing the TextView’s buffer by calling its clear() function, and zeroing out all password memory.

However, today’s developers generally do not share the same level of awareness of security practices. As found in our analysis, many apps simply send plaintext passwords to the network, or save them to local disks. A recent survey also confirms that developers think of functionality first, and regard security as a secondary task [61]. Therefore, we believe that, until a more prevalent awareness of password security forms in the Android community, it is more effective to develop a general, easy-to-use solution to avoid password retention.

We achieve this by designing and implementing KeyExporter, with two explicit design goals: proactive security, and usability. KeyExporter proactively contains passwords internally, and only exports password-derived keys to developers; it also securely manages password memory. Moreover, it offers developers simple APIs that are intuitive to use: existing studies show that if security APIs are complicated, developers tend not to use them at all [74, 75]. KeyExporter is designed with the understanding that usability and simplicity can effectively promote security [76].

KeyExporter Design. We start by identifying common patterns of password usage in today’s apps. Figure 3.4a shows the patterns for the 19 apps that we have analyzed, which are presented in the form of state transitions. We have omitted apps that rely on OAuth for authentication, as they do not directly manage passwords. The edges in this graph indicate
(a) The password usage patterns for 19 apps that we studied, in the form of state transitions.

(b) The password usage pattern KeyExporter aims to achieve.

(c) KeyExporter further encapsulates the states containing passwords.

Figure 3.4: The password usage patterns in the studied apps, as well as the pattern and encapsulation that KeyExporter achieves.
the flow of passwords in an app, and the nodes represent the states that a series of transitions can reach. Each node also contains a count, which represents the number of apps that reach this state following a particular transition path. For instance, all 19 apps implement a form of `onClick()` method, which is the starting point for authentication when a user clicks a login button.

We make several high-level observations on the usage pattern after `onClick`. Most apps (18 of them) directly get a `String` object from the widget to store passwords, and only one app correctly uses a `char` array. Afterwards, all of these apps perform a transition that we label as `use(pw)`, which either passes the data as a parameter to a function, or sends it to a different process via IPC, or checks its strength. We noticed that, apart from checking the password strength, all other “uses” are simply unnecessary password propagation. After this, 11 out of the 19 apps directly send passwords to the network instead of deriving keys from them. Although this percentage may not be representative for all Android apps (many of the tested apps here are sample apps), they do serve as further confirmation of the findings on insecure password usage in other studies [61]. Finally, the doubly-circled nodes in this graph correspond to the three common password sinks: networks, files, or cryptographic libraries that use passwords to derive keys. Furthermore, it can be noticed that two out of three sinks (i.e., sending passwords to the network or storing them in files) are insecure. These use patterns should be prevented, because they only lead to more password exposure. One should always use “key stretching” [56] to derive a strong key from the password and perform authentication with the key instead.

Therefore, our design of KeyExporter specifically focuses on preventing the possibility for passwords to be sent to the network or stored in a file, but rather encapsulates passwords internally and only exports password-derived keys. Figure 3.4b shows the intended usage of passwords enforced by KeyExporter: it always starts by a key derivation, and contains
passwords within the first three states (shown using blank circles); the rest of the states do not contain any user passwords (shaded circles). Figure 3.4c further abstracts the states that contain passwords into a super state, which shows the final design of KeyExporter. In current apps, developers need to obtain passwords from a widget, generate derived keys, and then manually clean up passwords; but with KeyExporter, a developer can simply call `getDerivedKey()` to obtain derived keys without ever accessing the raw passwords. All passwords are automatically zeroed out by KeyExporter.

**KeyExporter Implementation.** When an app developer uses KeyExporter, she would extract the needed key materials instead of the actual passwords from the widget. KeyExporter currently supports HMAC, PBKDF2, and scrypt as key derivation functions, although it could be easily be extended with others. We picked these functions because some of our test apps use HMAC and PBKDF2, and Android’s own device authentication uses memory-hard functions (MHFs) such as scrypt. In addition, we have also implemented support for the Secure Remote Password (SRP) protocol [66], which runs a password-authenticated key agreement (PAKE). Regardless of which method a programmer uses, KeyExporter prevents the spread of plaintext passwords, with API support for secure alternatives instead.

Figure 3.5 uses code segments to demonstrate how an app could use KeyExporter. Figure 3.5a is a simplified version of a sample app, which uses HMAC-based authentication instead of directly sending passwords; as such, it represents a more secure practice than the other sample apps. However, we can see that this app does not clean up passwords properly. Figure 3.5b shows the code after integrating the app with KeyExporter. In this version, the app no longer has direct access to the password. Instead, she could invoke a function to derive an HMAC based on the password for authentication. The `init()` call also clears the passwords from the widget and from memory. Figure 3.5c shows another
OnLogin:

```java
final String id = idUI.getText().toString();
final String pw = pwUI.getText().toString();

// generate HMAC
mac = Mac.getInstance("HmacSHA1");
key = new SecretKeySpec(pw.getBytes(), "HmacSHA1");
mac.init(key);
hash = mac.doFinal(serverRandom);

// send packet to server
sendResponse(id, hash);
```

(a) Original code for authentication using HMAC.

OnLogin:

```java
final String id = idUI.getText().toString();
HMacKeyExporter auth = (HMacKeyExporter) pwUI.getKeyExporter("HmacSHA1");

// generate HMAC
auth.init(); // cleanup happens here
auth.update(serverRandom);

// send packet to server
sendResponse(id, auth.getKey());
```

(b) Integration with KeyExporter using HMAC.
OnLogin:

```java
final String id = idUI.getText().toString();
SRPKeyExporter auth =
    (SRPKeyExporter) pwUI.getKeyExporter("SRP");

auth.init()  // cleanup happens here

// send ID, client public A to server
sendStep1Req(id, auth.getA());
...

// set received salt, server public B
auth.setStep1Response(s, B);

// send proof of session key to server
sendStep2Req(auth.getM());
```

(c) Integration with KeyExporter using SRP.

Figure 3.5: Integration with KeyExporter is easy.

Example based on the SRP API provided by KeyExporter, which uses a variant of Diffie-Hellman key exchange. As we can see, KeyExporter can achieve better security with more concise code.
3.7 Evaluation of SecureWidget and KeyExporter

We now report results from our experimental evaluation of SecureWidget and KeyExporter. We focus on three key questions:

- How effective can our solution fix password retention?
- How much code change does our solution require?
- How much development effort does a fix require?

3.7.1 Methodology

Similar as before, we have tested the effectiveness of our fixes for the Android framework (Problems #1-#4) and for the Android apps (Problems #5-#7). For each app $A$, we conducted two experiments:

- Running the original app on a modified version of Android that uses SecureWidget (written as $A'$).
- Running a modified version of the app that is integrated with KeyExporter and SecureWidget (written as $A^\dagger$).

Integration of KeyExporter with an existing app requires code modification, which is additional amount of work. In order to integrate with KeyExporter, one needs to identify the entry point for authentication, initialize KeyExporter with the desired cryptographic protocol, and replace all password usages with the derived key. We’re assuming that the relevant server-side code is already present, but for many app authors, this would represent an additional burden.

3.7.2 Effectiveness

Table 3.4 shows the results of our experiments, which we now summarize.
#1: Basic Password-based Authentication Applications. Sample app 1 uses passwords for authentication. As we can see from Table 3.4, the original app contains a large number of passwords. But the number is reduced roughly by half after using SecureWidget. The remaining password instances, which live in the app memory, completely disappeared after integration with KeyExporter. The results confirm that our solution effectively solves Problems #1-#7. Also, the result for the Sample app 1' confirms that password retention cannot be solved only by using an improved password entry widget.

Integration with KeyExporter only required changing six lines of code in the original app. More specifically, three lines of code were applied to the registration method, and three other lines to the login method. These changes replaced invocations of `getText().toString()` into `getKey()`, which is the method provided by KeyExporter for an app to retrieve credentials from `TextView`.

The original app sends raw passwords and usernames to the server for authentication. The server computes a hash of the password, and stores the hash in a local database (upon registration), or compares the hash with the local entry (upon authentication). We have modified the app to send the hash of the password instead, and removed the logic for the server to hash the password locally. We have fixed this sample app using less than one man hour.

Obviously, while this process hides the passwords, an adversarial capture of the hashed password is just as vulnerable as the capture of the original plaintext password. We next consider better protocols without obvious replay attacks.

#2: Challenge Response Authentication Applications Sample app 5 uses a challenge/response authentication. When running app 5', which is the version that uses SecureWidget, the number of passwords also gets reduced by about half. Integrating the app with KeyExporter further eliminates the rest of the passwords. After applying our fixes, app 5' is
<table>
<thead>
<tr>
<th>Category</th>
<th>Application</th>
<th>Version</th>
<th>Login</th>
<th>BG</th>
<th>Youtube</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Sample app 1</td>
<td>Original</td>
<td>25</td>
<td>24</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Basic</td>
<td>Sample app 1'</td>
<td>Enhanced OS</td>
<td>14</td>
<td>13</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Basic</td>
<td>Sample app 1†</td>
<td>+6 LoC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cha./Resp.</td>
<td>Sample app 5</td>
<td>Original</td>
<td>21</td>
<td>20</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Cha./Resp.</td>
<td>Sample app 5'</td>
<td>Enhanced OS</td>
<td>11</td>
<td>9</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Cha./Resp.</td>
<td>Sample app 5†</td>
<td>+5 LoC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OAuth</td>
<td>Yelp</td>
<td>Original</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OAuth</td>
<td>Yelp'</td>
<td>Enhanced OS</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Standalone</td>
<td>passwdSafe</td>
<td>Original</td>
<td>13</td>
<td>13</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Standalone</td>
<td>passwdSafe'</td>
<td>Enhanced OS</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Standalone</td>
<td>passwdSafe†</td>
<td>+50 LoC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>System app</td>
<td>SystemUI</td>
<td>Original</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>System app</td>
<td>SystemUI†</td>
<td>+200 LoC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PAKE</td>
<td>Sample app 1&lt;sub&gt;srp&lt;/sub&gt;</td>
<td>+110 LoC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.4: Our proposed fixes, SecureWidget and KeyExporter, can successfully address the password retention problem. A' is the version of app A running on Android framework with SecureWidget; A† is the version of app A integrated with KeyExporter and SecureWidget. Since Yelp is not open source, we were only able to apply SecureWidget, but not KeyExporter.

actually 16 lines shorter than the original app. This is because the original HMAC-based authentication protocol gets replaced by a simple invocation of KeyExporter. The source code has about 800 lines of code, and required about one man hour to fix this app.
#3: OAuth Authentication Applications

OAuth-based apps, such as Yelp, present a challenge for us to repair because neither Yelp nor its OAuth provider apps, like Facebook, are open source. Therefore, we were only able to test the password reduction on our patched Android that uses SecureWidget, which we can inject into other apps, as opposed to Key-Exporter, which requires app code changes. In this case, Yelp uses Facebook’s OAuth service, which retains much fewer passwords in memory than other apps. Nevertheless, password retention still occurred, and our SecureWidget reduced the password instances somewhat, but not to zero.

#4: Password Managers

Next, we tested the popular password manager, passwdSafe. As discussed before, its codebase reflects many good security practices, such as managing passwords using reference counts. Nevertheless, the developers failed to remove all passwords after authentication, as our results in Table 3.4 show. After integrating this app with SecureWidget and KeyExporter, all password instances disappeared immediately after authentication was complete.

In this case, the engineering challenges are that the app has 38,000 lines of code in 160 files, and we are not familiar with its source code. However, our modifications can be applied in a straightforward fashion. Figure 3.6 shows our core changes by comparing the original code (3.6a) and our modified code (3.6b). We first located the code for the login file using a simple pattern match, and identified \texttt{R.id.ok} to be the starting point. We integrated this method with the initialization routine for PBKDF2. For the rest of the codebase, there is only one place that uses the master password; therefore, we replaced this usage with the hash value derived by KeyExporter. In total, our fix used 50 lines of code, although the whole process took several man hours to complete and test. Most of the time was spent understanding the codebase to find the entry code for the login. Nevertheless, our success with fixing passwdSafe demonstrates that integrating our fixes with a sizable program is
straightforward.

**#5: System Processes.** Next, we tested lockscreen processes (SystemUI and LocalSettingsService) which leave PINs in memory (Table 3.4). We found that the starting point for authentication is in the KeyGuard module in the SystemUI process, so we generated derived keys using scrypt in this module and deleted passwords right away. We found more than ten methods to which the passwords are passed, so we fixed all these methods. The result for SystemUI† in Table 3.4 shows that the password retention problem was solved completely after the fixes: right after unlocking the phone, all passwords disappeared.

This fix took about 200 lines of code, most of which modified the function prototypes. Although we are familiar with the Android codebase, applying this fix took three days of work for one developer. This is because we had to go through the codebase to trace how the system processes use the PIN password. Nevertheless, we were able to apply the same types of fixes to solve the problem.

**#6: Password Authenticated Key Exchange.** Next, we evaluated our PAKE support. We cannot find secure sample apps that use PAKE protocols such as SRP. Thus, we decided to improve a naïve app with the secure protocol, allowing us to measure the effectiveness of our system as well as the time effort required to make the changes. Sample 1 can solve password retention by avoiding the use of passwords as Sample 1†. But, still, in this app, the derived secret is simply a hash of the password, presenting security vulnerability for man-in-the-middle attacks. We modified this app to use the SRP protocol in KeyExporter. This requires an additional change of 110 lines of code: 30 in the app, and 80 lines in the server side to support SRP. For the adaptation, Diffie–Hellman like key exchange routine should be implemented, calculating various cryptographic values. Note that only 30 lines are added in the client code for implementing both registration and login routines. Sever requires additional modification on the database because it never sees user password or
// OnClick

case R.id.ok: {
    if (itsYubikeyCb.isChecked()){
        ...
    }
}

// Open Database

Owner<PwsPassword> passwd =
    new Owner<>(new PwsPassword(passUIitsPasswordEdit.getText()));
    ...

(a) Original authentication code in passwdSafe.

// OnClick

case R.id.ok: {
    keyExporter = (PBKDF2KeyExporter)
        itPasswordEdit.getKeyExporter("PBKDF2");
    keyExporter.init();
    if (itsYubikeyCb.isChecked()){
        ...
    }
}

// Open Database

Owner<PwsPassword> passwd =
    new Owner<>(new PwsPassword(keyEXporter.getKey()));
    ...

(b) Integration with KeyExporter using PBKDF2.

Figure 3.6 : Integrating passwdSafe with KeyExporter.
hash, instead, they have to store a crypt verifier and a salt value.

Sample 1 shows the result of the fix: as before, all passwords are successfully cleaned after login. Given our simplistic sample app, it took a couple of hours to apply SRP (client and server-side) and make sure everything worked. Of course, not all developers will be comfortable with SRP. Even when the client-side code is easy to integrate, the server-side code might be much more complicated, or might have its own closed-source legacy components that a developer cannot easily fix. Nonetheless, our evaluation shows that KeyExporter correctly manages client-side plaintext passwords in memory and can help developers follow stronger cryptographic practices.

3.7.3 Summary

To summarize, there are four key takeaways from our evaluation. First, our solutions are effective in solving password retention in Android, as they can successfully remove all passwords from our tested apps (with the exception of Yelp, which is not open source). Second, the lines of code in a fix are app-specific, but generally speaking, they are relatively small. Third, the modification can be done in a systematic manner by following the principles of the fixes in Section 3.6, and modifying an app takes hours in most cases. Lastly, fixing an app “correctly” often requires the adoption of a cryptographically strong protocol like SRP. Our proposed solution not only helps reduce password lifetimes, but also helps developers migrate to these stronger protocols, which they should be using anyway.
3.8 Discussion

Other threat models. As we discussed, our primary goal is to protect user passwords against an adversary that can perform memory disclosure attacks, which is a common threat model in many previous studies \[18, 27, 46, 49, 50, 51, 62\]. Our solution successfully clears passwords right after login, providing an effective defense against such attackers. Moreover, the design of KeyExporter can further prevent passwords from being propagated to the rest of the codebase, so it prevents the possibility for passwords to be leaked to the network or stored in files.

Nevertheless, memory disclosure attacks are not the only way in which an adversary can compromise user passwords. If an attacker can compromise an Android phone and gain root privileges, it is possible that she might be able to perform real-time monitoring of touchscreen activities to capture the password. This is outside the scope of this current work. A possible defense, however, may be to leverage TrustZone for protection, similar as how the Android framework protects fingerprint data from attackers.

Limitations. Our solution with KeyExporter requires code changes in applications, though the changes are typically quite small. If we do not have the source code for an app (e.g., Yelp), then we can only apply SecureWidget to reduce the number of password copies, but not KeyExporter, which needs to be integrated with an insecure app at the source code level. Preventing password retention without modifying the app seems fundamentally difficult, as is also noticed by Chow et. al \[48\]. Out of the seven root causes that we have discovered, three are due to developer mistakes. As such, they can only be addressed in an app-by-app fashion by applying the fixes. The fixes for the Android framework, on the other hand, are a one-time change that can benefit all applications running on top.

Usability. Our fixes to the Android framework with SecureWidget do not require any further changes to upper-layer applications. KeyExporter does require changes to the apps,
but its design follows the usability principles identified by recent studies. As developers prioritize functionality over security [61], we have designed KeyExporter to be not only more secure but also provide a rich set of additional functionalities, such as hashing and PAKE. We believe that providing these functionalities can further attract developers to integrate their apps with KeyExporter. Moreover, studies have found that developers do not use security APIs at all if they are too complicated [74, 75]. KeyExporter follows this principle that simplicity promotes security [76], and exports key materials in a similar fashion as how existing widgets export passwords. The code changes to integrate with KeyExporter are small, which lowers the effort for an app to be integrated with our solution.

**Centralized security management.** By shifting the responsibility of password management from app developers to KeyExporter and SecureWidget, we relieve the developers from having to reason about password security manually app-by-app, and we can harden the system from password misuse by the app developers. Needless to say, the design and implementation of SecureWidget/KeyExporter need to be secure; otherwise, all apps that are integrated with them would become vulnerable again. As KeyExporter can localize the reasoning of passwords to this one component, such “centralized” security management is similar in spirit to how TLS eases the burden for developers to implement secure communication, or how OAuth centralizes authentication by managing user credentials in a small number of trusted service providers.

**Reporting security flaws.** Our analysis has revealed many problems in the Android framework and popular apps. Some of them happen because developers have not exercised any secure practices in the app. For instance, in the case of lockscreen PIN passwords, the codebase does not reflect an awareness of password retention, as no attempt of secure management can be seen in the code. Other incidents happen in apps that show many signs of applying secure password management in the codebase, but they have not thoroughly
identified all cases of retention (e.g., passwdSafe). We have disclosed these security problems to the respective companies that developed the vulnerable apps. In particular, we have contacted Google to report the problems in the AOSP keyboard, the lockscreen processes, and the Android widget implementation. We also contacted the keyboard app developers and the OAuth team at Facebook, because these apps are integrated with many other third-party apps. Finally, we have reached out to the password manager developers with proposed fixes. Also, we provide public code samples explaining to developers how they can harden their apps using our KeyExporter and SecureWidget systems.

**Keyboard and password-management apps.** Keyboard apps have a high burden for security. If a keyboard app were deliberately malicious, it would be able to record and exfiltrate everything a user types. This suggests that keyboard apps should face heightened scrutiny, both for the specific issue of password retention that we highlight in this chapter, as well as for a variety of other potential threats to their users. While it’s interesting to consider this as a research challenge, treating this as simple information flow analysis would falsely flag every text-predictive keyboard that tries to maintain a custom “cloud” dictionary for its users. The number of popular keyboard apps is small enough that Google might simply require third-party security auditors to examine and approve them individually before allowing them into the app store.

A similar level of scrutiny seems appropriate for password-management apps, for all the same reasons. Users cannot readily distinguish how strong or weak these apps might be. As with keyboard apps, an automated analysis would likely stumble over the “cloud” features that allow more sophisticated password management apps to synchronize passwords across a user’s various computers and mobile devices. Third-party security auditors again seem like a suitable solution.

**What about static analysis?** Google regularly adds static analysis features to its Android
Studio development tool to highlight coding practices it considers undesirable. If Android were to adopt our alternatives to the password entry widget, or something like them, then they could simply deprecate the password features of the default TextView widget, generating warnings that every Android developer would see every time they compiled a program. Of course, developers will always manage to find insecure ways around such warnings, but a campaign by Google to improve developers’ code could certainly help nudge its app ecosystem toward better password-management practices.

Future work. Our current analysis is targeted at user passwords. Going forward, we believe that looking at credentials derived from passwords is also an interesting topic. Does Android also retain cookies and authentication tokens? Are they vulnerable to memory disclosure attacks? If so, can we apply similar fixes, or do we need to develop new solutions? Moreover, we also plan to investigate the feasibility for an attacker with root privileges to monitor keystrokes and capture passwords, and explore potential mitigations that protect passwords using TrustZone.
3.9 Related Work

3.9.1 Protecting sensitive data

Insecure data deletion is one of the fundamental causes for data exposure. Researchers have developed many solutions to address this. Chow et al. handle this by secure deallocation [48]. Pridgen et al. aim at reducing encryption keys retained in the Java heap for desktops and servers [22], Dunn et al. use ephemeral channels where data will be securely erased after a session finishes [77], and Lee and Wallach study the retention of TLS secrets in Android memory [62]. Different from existing work, our work focuses on the study of password retention, and proposes effective fixes to address this problem.

Another line of research looks at storing sensitive data in secure locations, instead of removing data from insecure locations. Example secure locations considered by existing work include the cloud storage [46], registers [49], and TrustZone [51]. These proposals require an additional storage other than the main memory, so the hurdle for deployment is much higher; they also tend to cause higher performance overhead.

Researchers have also considered protecting sensitive data by detecting malicious application misbehaviors. Dynamic analysis techniques, such as data-flow analysis [70] and password-tracking [71], have been introduced to detect data leakage from applications. Static analysis techniques [67, 68, 69] have also been used to detect malicious behaviors. These approaches facilitate an automated reasoning about insecurity, but they are limited to performing analysis to either the C layer or the Java layer. In contrast, we use a manual, exploratory approach to analyzing password retention, which covers the entire software stack.
3.9.2 Memory Forensics

In terms of memory acquisition techniques, Sylve et al. are the first ones to suggest a technique for capturing the physical memory of Android devices [32]. Yang et al. design an acquisition technique when the Android device is in firmware update mode [78]. We have used the system developed by Sylve et al. to perform our study.

In terms of memory analysis techniques, signature-based framework [35] have been widely used by researchers to analyze memory. Various techniques have also been proposed to recover data structure from memory dumps using static analysis [39], dynamic analysis [37], and probabilistic analysis [38]. Memory forensics has been applied to the Android platform. A line of previous work has focused on extracting sensitive data from applications [18, 27, 40, 41]. Researchers have also looked at techniques to recover data beyond raw memory dumps, including the timeline of user activities [79] and GUI activities [42, 43].

3.9.3 Security flaws in Android apps

Existing work has studied security flaws in Android apps, and revealed that developers have misused TLS library [6, 7, 8], cryptographic APIs [9], OAuth protocols [10], and fingerprint APIs [11]. Reaves et al. analyze mobile banking apps, and report information leakage in these apps [80]. Recent usability studies have also looked at why developers make mistakes, by analyzing the patterns of misuse [61, 72, 75, 76, 81]. The motivation of our work is inspired by these studies.

Password managers have attracted much attention because they directly handle sensitive passwords. Fahl et al. reveal that many password managers are vulnerable to clipboard sniffing attacks [82]. Silver et al. report critical flaws in the auto-fill functionality [83], and Li et al. report problems in web-based password managers [84].
3.10 Conclusion

In this chapter, we have performed a comprehensive study on password retention in Android. We have found problems with the Android platform (e.g., the lockscreen process), as well as a wide variety of apps (e.g., password managers, banking apps). Furthermore, we have also identified the key root causes for retention and proposed fixes: SecureWidget, and KeyExporter. SecureWidget fixes the retention problems in the default TextView class on the Android platform, whereas KeyExporter can be integrated with Android apps and provide automatic, secure password management. Our experiments demonstrate that our fixes are effective on all tested apps, and that they only require a modest amount of code changes.
Chapter 4

**FlowPass: Understanding password misuse in Android**

### 4.1 Introduction

Many mobile apps, whether on Android or iOS, serve as a front-end for various remote services, many of which need their users to log in. Of course, many authentication mechanisms would be suitable for this [55], but password-based authentication is still the de-facto standard. While new mechanisms like fingerprint authentication enjoy special hardware support (e.g., well-defined APIs for invoking TrustZone [11]), password authentication lacks anything special from the Android framework [85]. Developers want something that’s portable across different mobile platforms and web browsers, so they go with what’s easy, and the resulting apps will vary in terms of their password handling practices.

Unfortunately, past findings paint a grim picture on how developers repeat mistakes on implementing necessary security features. Android developers have been found to misuse TLS [6, 7, 8], fingerprint APIs [11], cryptographic APIs [9], OAuth [10], and cloud APIs [86]. Recent studies have further revealed why and how inexperienced developers may implement security features poorly [61, 72, 76, 81, 87].

What about password misuse? In fact, security vulnerabilities due to password misuses are regularly discovered in real-world applications (e.g., CVE-2018-11505, CVE-2018-18698, CVE-2019-12820, CVE-2019-13098, and CVE-2019-13099). These findings are very concerning because password compromises have disastrous consequences, particularly with users reusing their passwords across different services [63, 64, 65]. Making mat-
ters worse, developers tend to store passwords in plaintext [61], so compromised servers can result in millions of usernames and passwords being leaked. While we cannot easily study the backend servers, we can certainly study the front-end apps. Can we systematically study Android apps to see how they use and misuse passwords?

To this end, we considered a variety of existing static and dynamic analysis techniques to find a proper systematic solution. However, no existing tool works out-of-the-box for analyzing password misuse. Static analysis approaches can reason about an app without running it, but they have problems in practice due to obfuscation and dynamic loading. For example, FlowDroid [67], AmanDroid [88], and DroidSafe [89], have practical problems in their analysis of real-world apps [90, 91, 92]. Alternatively, “lightweight” static analysis engines also exist, but they are typically customized for reasoning about only a few properties for specific problems [6, 9, 10, 11]. On the other hand, dynamic taint analysis is powerful and identifies data leak during execution [70, 93, 94, 95]. However, these systems aren’t particularly portable, so we would have to use the much older Android versions that they support. And, of course, even if we could make them run, we would have a variety of false positives, since an encrypted value, derived from a password, is still “tainted” by that password, even if the cryptography is sound. Furthermore, code obfuscation tools might deliberately create false dataflows, creating corresponding false-positive indications of password leakage.

In order to understand the misuse of passwords, our system must satisfy three goals. First, our tool should be effective on complex real-world apps, which may have a large codebase and complex interactions with third-party libraries. Second, our tool should be informative. Upon detecting problems, it should produce detailed diagnostic information about its findings; for example, it must distinguish passwords that have been encoded (base64 or otherwise) rather than encrypted. This would be helpful for developers to con-
firm and understand their problems. Third, the tool should not be tied to a specific Android version or runtime, as updates to these platforms happen very often. Portable tools are obviously preferable.

With these goals in mind, we designed and implemented a new dynamic analysis framework, FlowPass, for analyzing apps’ password handling practices. Instead of performing a fine-grained dynamic analysis, tracking taints at the instruction level, FlowPass taints common Java objects that may be used for storing passwords, such as String objects, char arrays, and byte arrays, trading off completeness for efficiency. FlowPass propagates taint information at the function call level when well-defined APIs are invoked on these data types. This procedural-level propagation allows FlowPass to preserve Java semantics and provide a comprehensive view of password flow during execution, and to distinguish cases where passwords are propagated or leaked in reversible (e.g., plaintext, simple encoding) or secure (e.g., encrypted, hashed) forms. Also, FlowPass is relatively easy to port to future Android versions since enhancing these data types mostly requires localized instrumentation to Java core libraries, which are stable across Android versions and runtimes.

We did make a number of compromises. Android supports x86 and ARM native code as well as its Java-based runtime system. We chose not to track native code password use. Similarly, we make no particular attempt to deal with the complexities of code obfuscation. These compromises will limit our ability to analyze every Android app.

We have evaluated FlowPass on popular Android apps, and obtained insights on password misuse and identified unknown security problems. We downloaded 1,903 popular Android apps, finding that 182 use password-based authentication. FlowPass found 13 previously unknown security bugs in these apps, which have been installed more than one million times, including DoorDash, Orbitz, and Infinite Design. FlowPass has identified three common patterns of password misuse, and can output provenance traces for diagnos-
tic analysis. FlowPass also reveals interesting differences between popular apps in Google Play and comparably popular apps from stores in China. Google Play apps tend to use more naïve handling of passwords, sending plaintext passwords to the network, assuming that TLS will protect them. (We discuss why this is undesirable in the next section.) A higher percentage of Chinese apps encrypt passwords additionally before transmission.

After presenting the problem statement in Section 4.2, we make the following key contributions:

- Comparative analysis of static and dynamic approaches, which provides insights for potential new research directions (Section 4.3);
- An efficient and informative dynamic taint analysis design, specialized for analyzing password misuse in Android apps (Section 4.4);
- FlowPass, our implementation of this technique;
- Evaluation with top apps, finding previously unknown security bugs (Section 4.5);

We then discuss several issues in Section 4.6 and related work in Section 4.7 before concluding in Section 4.8.

### 4.2 Problem Statement

In this section, we describe the threat model that FlowPass considers, define password misuse, and discuss challenges.

#### 4.2.1 Threat model

We’re starting with a straightforward threat model, where attackers can capture any data transmitted outside of the app. For example, attackers can sniff traffic in public WiFi. Communication via WPA2-encrypted WiFi, for example, can still be captured [24]. Also,
many other installed apps can look at shared files or log data, so any raw passwords written to files or to the Android log constitute leaked passwords. Attackers can even obtain sensitive data even after an app is uninstalled [96]. Therefore, passwords should always be encrypted whenever sent out of the process.

Recent studies assume even more powerful adversaries who can compromise the operating system, or can fully access the memory of an app’s process [62, 97, 98]. In this situation, plaintext passwords should be kept in memory no longer than necessary to minimize their exposure [46]. We are not assuming this strong of an adversarial model, although it’s still a good practice for developers to minimize the presence of plaintext passwords.

4.2.2 Categorizing password misuse

Based on the threat model above, we define the following three cases as password misuse.

Case #1: Exposure of plaintext passwords. This is a fundamental security practice. If an app sends out plaintext passwords to the network, logs them via logging facilities, or stores them in a file, these are considered misuses. Figure 4.1a shows an example of this misuse. It sends passwords to the network using the Volley library. Despite the simplicity of this example, recent static analyzers fail to detect this leak, as we will explain later.

Case #2: Exposure of reversible forms of passwords. User passwords may experience a variety of transformations. We define that a transformation is reversible if an attacker can derive the original password without knowing any shared secret. By this definition, encryption is irreversible, but encodings or lossless compressions are reversible transformations. If an app exposes reversible forms of passwords to the external environment, this is considered to be a misuse. As an example, consider the misuse shown in Figure 4.1b; the app encodes the password using Base64, compresses it, and sends it out using the OkHttp library.
public void login(String id, String pw) {
    // create request packet with password
    LoginRequest request = new LoginRequest(id, pw, responseCallback);

    // add the request into queue, which will be sent later
    RequestQueue queue = Volley.newRequestQueue(thisActivity);
    queue.add(loginRequest);
    ...
}

(a) Sending plaintext credentials using Volley library.

static Future<Object> login(OkHttpClient client, String id, String pw) {
    // Base64 encoding
    String credentials = id + ":" + pw;
    final String b64Cred = "Basic " + Base64.encodeToString(credentials.getBytes(), Base64.NO_WRAP);

    // send request using OkHttp API
    Request request = new Request.Builder().url(LOGIN_URI).
        .addHeader("Authorization", b64Cred)
        .addHeader("Accept_encoding", "gzip")
        .build()

    Response response = client.newCall(request).execute();
    ...
}

(b) Sending compressed Base64 credentials using OkHttp library.
public void login(NetworkManager net, String id, String pw) {

    // receive challenge value from the server
    byte[] serverChallenge = net.recvChallenge();

    // generate hash from password and challenge
    mac = Mac.getInstance("HmacSHA256");
    SecretKey key = new SecretKeySpec(pw.getBytes(), "HmacSHA256");
    mac.init(key);
    mac.doFinal(serverChallenge);

    // send response value to the server
    String reply = net.sendResponse();
...
}

(c) Sending hashed password using a challenge-response scheme.

Figure 4.1: Motivating examples.

**Case #3: Sending plaintext passwords via TLS.** Using TLS is much better than sending raw passwords in the open. However, sending a plaintext password through TLS is inadequate against modern threats [99]. TLS is plagued with vulnerabilities [100, 101, 102, 103, 104, 105]; an adversary might compromise TLS or decrypt its ciphertext by SSL stripping, command injection attacks, attacking the underlying cryptography, or stealing private keys (see, e.g., [106, 107]). A recent study shows that 10% of TLS servers have known vulnerabilities [99]. Therefore, an app should not delegate plaintext passwords to TLS as the only means of protecting them [99]. Interestingly, for the developers, there is very little difference in sending data via TLS or a vanilla TCP connection if they use common libraries like Volley. Figure 4.1a shows an example that sends plaintext passwords to the network using
the Volley library, and it will send passwords via TLS if its destination URL starts with “HTTPS” instead of “HTTP”.

4.2.3 Secure password usage

In contrast to the password misuses above, a better practice would be to derive cryptographic values from passwords and using those derived values for authentication. Figure 4.1c shows a challenge-response protocol, where the app hashes the password with a challenge from the server, and sends the hashed value to the network. In our study, this is considered to be secure.

Attentive readers would notice that this example uses SHA256 hashing, and ideally, developers need to use stronger hash functions such as scrypt [59]. Preferably, apps will use a password-authenticated key exchange (PAKE) protocol, such as Secure Remote Password (SRP) [66]. We note that FlowPass does not aim to analyze the strength of encryption, or further identify misuses of cryptographic APIs. Studies on cryptographic misuses are complementary to our work [9, 108, 109]. We’re interested in discovering practices that are clearly inadequate.

4.2.4 Challenges.

To understand password misuse, a systematic approach must address the following challenges:

- **Identify password flow in apps with various network libraries.** We found that recent powerful static analysis engines [67, 89, 88] fail to detect password leaks in the simple case of Figure 4.1a. Network libraries are complex, involving threading, asynchronous calls, and queue handling, creating complexities that hamper static analysis [110]. Any static analysis approach built with current tools would need to
create external models of the networking libraries’ behavior to hide their complexities.

- **Distinguish the four different usages.** Having identified password flows, we still need to determine if the exposed data is reversible or irreversible. This is especially challenging for a dynamic tainting approach, because they propagate taint information regardless of reversible or irreversible transformation. We could work around this by embedding knowledge of common APIs for cryptography and compression into the taint tracking, but these would miss any external libraries or homebrew approaches that an app might include. Also, there may be significant runtime overheads to tracking tainted data as it flows through an encryption or compression system.

### 4.2.5 Design goals

We aim to design a system that identifies the above misuses while satisfying three design goals:

- **Effective.** First and foremost, the system should be able to handle complex, real-world apps. It should be effective to find password misuse where it exists, running efficiently and with a tolerable number of false alarms.

- **Informative.** In addition to detecting problems, FlowPass should also generate useful information for the app developer to confirm and understand their problems, verifying when the bugs are repaired. We require more feedback than a simple binary decision, “good” vs. “bad”.

- **Portable.** FlowPass should also work across Android versions and runtimes. If the analysis relies on heavy instrumentation at the bytecode-level or low-level customizations of the Dalvik or ART runtimes, then the tool will be highly dependent
on that specific versions of Android. A tool that requires minimal changes will likely be more easily ported to newer Android versions and thus have a longer shelf life.

4.3 Analyzing the Design Space

First, we have surveyed possible approaches to analyzing password misuse in terms of our challenges and design goals. Below, we categorize potential approaches in two dimensions: a) whether they use static or dynamic analysis, and b) whether they are general-purpose or specialized for a class of problems.

4.3.1 General-purpose static analysis

The first class of candidates use static analysis to track data flows in Android apps. We look at three popular static analysis frameworks: AmanDroid [88], DroidSafe [89], and FlowDroid [67]. They have strong theoretical underpinnings [111]; these tools also tackle additional challenges for Android apps by modeling Android’s unique lifecycle. They require an analyst to define source APIs that capture sensitive data (e.g., `TelephonyManager.getDeviceId()`), and sink APIs that define undesirable places where sensitive data might end up (e.g., `FileOutputStream.write(char[])`). After these annotations are specified, these tools detect data leaks from any sensitive source to any sink, all without running the app.

**Pros and cons.** These tools are powerful and flexible to detect various types of information leakage. However, this can be a downside in terms of the precision and complexity of apps they can handle. Previous studies [90, 91, 92] have revealed that the tools suffer from precision problems. Specifically, these tools can achieve high accuracy with small apps created as microbenchmarks in DroidBench [67]. However, when they are provided with mutated test cases, the false positive rates grow to 4.1% for DroidSafe, 48.7% for FlowDroid, and 73.1% for AmanDroid [55]. Also, they have difficulty handling complex
apps. For example, DroidSafe can only analyze two out of 30 of the tested apps [91]; FlowDroid fails to analyze all top 10 banking apps [45].

As a microbenchmark, we have adapted the above tools by adding network APIs in the sink list and testing whether they can detect password leaks to the network. We ran the example in Figure 4.1a for this purpose. AmanDroid and FlowDroid did not raise any warning on the password leak, and DroidSafe crashed due to Out-Of-Memory errors after running for 30 minutes on a server with 32 GB RAM. We found that this is because the example snippet interacts with network libraries, which are usually quite complex. The Volley library in the example internally calls OkHttp and other libraries, going through a deep call chain before reaching the actual socket calls. These libraries are also built with complex constructs for threading, asynchronous events, request queues, and buffer management. Extending these tools to support these advanced features would require a significant amount of research and engineering efforts [90].

4.3.2 Specialized static analysis

The second class of candidate approaches customize static analysis techniques to target a specific set of problems, achieving a much more lightweight analysis. For instance, MalloDroid [6] detects misuse of TLS APIs, CryptLint [9] detects misuse of cryptographic APIs, Bianchi et al. [11] detects misuse of fingerprint APIs, and LeakScope [86] detects misuse of cloud APIs. Despite these seemingly discrepant goals, these tools share a similar workflow and approach. First, the authors must manually define insecure patterns that the tools should look for. For instance, CryptLint defines six common rules in misuses of cryptographic APIs, e.g., “do not use ECB mode for encryption”; LeakScope analyzes method calls in cloud APIs and checks whether or not they pass the root key as arguments. These authors then developed static analysis tools that specifically checked these known-bad pat-
terns. This can be achieved using static program slicing [112], where the tool starts from invocation of target APIs, looks at the program backward, gathering all the statements that contribute to the target.

**Pros and cons.** Compared to general-purpose static analysis techniques, this second class of tools often only need to perform localized analysis, e.g., for specific API uses, so they are efficient and scalable. For example, LeakScope can handle even millions of real-world apps. The downside is they can only detect the issues for which they were specifically built, so might miss slightly different variations on the same issues. Also, these approaches can have significant false-positive rates. For example, LeakScope show that 92% of tested apps use Google Firebase, and only 4% of apps use Amazon AWS, while we know that AWS is a market leader for cloud services. Perhaps the wide variety of network libraries make it harder to identify apps which use AWS. Also, any static analysis tools will have difficult handling dynamic loading or code obfuscation (e.g., 3,386 out of 15,134 apps failed in CryptLint). More relevant to our setting, these tools are highly specialized for their respective problems. None can detect the sort of password misuse that we wish to discover, so we would need to build our own system.

4.3.3 General-purpose dynamic analysis

The next two classes of techniques use dynamic instead of static analysis. They attach taint bits to sensitive data and propagate the taints across the system during execution. Supporting taint tracking, of course, requires instrumentation at various levels. Many dynamic analysis tools exist for general-purpose taint tracking. For instance, DroidScope [95] modifies the emulator, TaintDroid [70] instruments the the Dalvik interpreter, TaintART [93] and Artist [113] changes the ART compiler, and Malton [94] uses Valgrind’s instrumentation [114].
Pros and cons. A key advantage of these approaches is that they sidestep the problems of dynamic loading and obfuscation, making it easier to support a larger number of real-world apps. However, several notable limitations exist. First, we noticed they all suffer from severe maintenance problem unlike static approaches. For instance, TaintDroid only supports Android 2 and 4, and DroidScope supports Android 4; these versions are both outdated. TaintART is not maintained since the first release for Android 6, and Artist stops at Android 8. We have attempted to port them to the latest Android 9, but we found that significant efforts would be needed, on par with those required for building a new tool. This is because such tools make intrusive changes to low-level features, such as the Dalvik and ART compiler; these features go through frequent changes even for minor revisions in Android.

Another limitation is that they suffer from the “semantics gap problem.” TaintDroid, TaintART, and Artist instrument and propagate taint bits at the instruction level, so they lack information about higher-level semantics that is needed for diagnostics. TaintDroid, for instance, generates outputs about taint source, process ID, and tainted buffers, but does not provide the provenance of the leak in terms of call graphs. Consequently, it is difficult for the app developer to use taint tracking information to confirm and understand how their app might be problematic.

The last problem is inefficiency, because they seek to track every kind of taint propagation. Malton and DroidScope, engineered to overcome the semantic gap problem, incur severe overhead in their efforts to reconstructing high-level semantics from the low-level instructions they observe. Malton, for instance, reports a slowdown of $30 \times$, and DroidScope reports a slowdown between $11 \times -32 \times$. And, we have tested an app with DroidScope which is released in open source. It fails in the login step due to network timeout.

To summarize, general-purpose taint tracking techniques are very powerful, but they
have portability issues, semantics gaps, and/or low performance. Since our goal in Flow-Pass is to specifically analyze password misuse, we could potentially do better by customizing the analysis.

4.3.4 Specialized dynamic analysis

The fourth and final class of techniques customize dynamic analysis for a specialized class of problems [115, 116, 117]. Existing techniques of this class target web applications instead of Android apps, and they are interested in detecting SQL injection attacks to database backends. They instrument JVM libraries to support taint tracking specifically for the Java String class and related classes like StringBuilder and StringBuffer.

Pros and cons. Since they highly customize the taint tracking logic for String objects, the analysis is very lightweight and efficient. Also, these techniques are potentially more portable across Android versions and runtimes. In terms of downsides, taint information will be lost whenever String objects are converted to other data types, such as char or byte arrays. In fact, this is a significant drawback for the purpose of password analysis, because network libraries usually convert String objects to array types before sending them out. A tool that builds upon this class of techniques would need to address precision and coverage problems.

4.3.5 Summary

Obviously, there is not silver-bullet solution that consistently outperforms others in all dimensions. In addressing our problems, existing static analysis techniques have problems analyzing complex apps, due to difficulty in handling code obfuscation and dynamic loading, whereas dynamic analysis techniques suffer from the trade-off between presenting useful diagnostic information and running efficiently, due in part to the costs of working
with low-level fine-grained taint propagation of all possible objects and values.

As a result of our survey, we have decided to take a dynamic approach, specialized to our specific problem. In the next sections, we will describe our design and engineering efforts and results from this approach.

### 4.4 FlowPass: Design and Implementation

The previous section shows that the fine-grained dynamic taint frameworks are either informative but slow, or fast but not informative. In this section, we present FlowPass that achieves efficient and informative taint tracking in a coarse-grained manner.

#### 4.4.1 Observations and design decisions

Our design principles are grounded on two key observations on how developers use passwords.

**Observation #1:** *Passwords are commonly stored in a few key data types.* Our first observation is that developers typically store passwords in String objects, char arrays, and byte arrays, instead of primitive types, such as individual char variables (e.g., one char variable per password digit). This makes it possible for us to very selectively add taint support for a few key data types, without having to taint every byte of memory. Our design decision, therefore, is to *manage taint tags only for specific Java object types* so that we can reduce the overhead of taint tracking.

**Observation #2:** *Password are typically handled with well-defined APIs.* Second, developers typically use passwords in their entirety, instead of manipulating them at the byte level or using fragments of them. Moreover, there exists a well-defined set of APIs that we would expect to be called on password data, such as storing them to disk, sending them...
to the network, or encrypting them. This observation helps us narrow the scope of taint tracking. Our design decision, therefore, is to *annotate well-defined APIs and propagate taints at the function call level.*

### 4.4.2 Design overview

Before delving into the details of our design, we explain more on the high-level architecture of FlowPass, as well as the implications of our design choices.

First, we have opted to store taint tags only for three types of Java objects: Strings, char arrays, and byte arrays. Our design stands in contrast to the recent trend of dynamic analysis techniques targeting malicious apps, which favor completeness over efficiency and aim to trace even implicit control flows [71]. FlowPass is able to adopt a new point in the design space because it focuses on analyzing *benign* apps to identify misuses, not intentional compromises. This goal allows us to apply aggressive optimizations while preserving effectiveness. For instance, FlowPass does not have to maintain fine-grained taint tags per byte; if an app performs a byte-level copy, say, \( B = A[1] \);, where \( B \) is a char variable and \( A \) is a tainted array, FlowPass will not further taint \( B \).

A second implication stems from our decision to propagate taints at function calls (e.g., `System.arraycopy()`). This leads to another layer of optimization compared to many existing systems, which propagate taints at the instruction level. For instance, TaintDroid [70] instruments core Dalvik instructions (e.g., `move-object`), and TaintART [93] instruments compiled instructions by the ART compiler (e.g., `HParallelMove`). Tainting at the instruction level favors completeness; since the instructions form a closed set, one only needs to instrument a limited number of operations (e.g., TaintART instruments nine instructions). In contrast, FlowPass needs to identify key APIs that manipulate arrays and Strings from an open set of framework libraries. In return, FlowPass can be more efficient and portable.
Figure 4.2: The architecture of FlowPass. Dashed boxes represent our modification or addition to Android Open Source Project (AOSP). We have further labeled the boxes as sink, source, taint storage, or provenance tracing. Dashed boxes without labels are modified for taint propagation.

Figure 4.2 shows an overview of the FlowPass architecture, where dashed boxes represent the components that we have instrumented for taint tracking. Concretely, there are four such components in the Android framework: Framework Base Core, Java Library Core, External Libraries, and ART Runtime. We have instrumented these components to store taint information and to propagate taints across function calls. In addition, we have introduced a new component—the TaintManager module—that provides utility APIs for taint tracking and provenance tracing. Below, we discuss these components in more detail.
4.4.3 Storing taint tags

The granularity of taint tags determines the performance and memory overheads of taint tracking. Whereas existing systems store taint tags for every single byte, e.g., in shadow memory [47, 118], tag maps [119], or adjacent to the main variables [70], FlowPass simply stores one bit of taint tag for every String, char array, and byte array.

**String objects.** Unlike strings in the HotSpot JVM, which are implemented in pure Java except for the one native method `intern()`, Android strings are a special entity. Android implements Strings using a combination of Java, C++, and assembly code. The Java implementation resides in the Java layer of the Android framework, and the C++ implementation in the mirroring component in the ART runtime; Figure 4.3 shows this division. More concretely, `java.lang.String` at the Java layer implements functions like `trim()` and provides interfaces to the C++ layer; core data structures and methods, such as `charAt()`, are implemented in the C++ `mirror::String` class. An invocation of a core method will be delivered to the ART runtime via JNI. This entangled design forces us to modify the ART runtime’s native string code to support taint tracking. To store the taint tag in a string, we borrow one bit from the `count` member variable in the `mirror::String` class. Android already borrows one bit from this variable to distinguish Unicode strings: if this bit is set, the string will have two bytes instead of one byte per character for Unicode encoding. The net effect of borrowing another bit is that the maximum string length is reduced from $2^{31}$ to $2^{30}$. In practice, Android memory limitations already forbid creating strings of such sizes. Moreover, the size of the string object remains the same, so the Android framework sees no additional side effects.

**Byte and char arrays.** These array types are also Java objects, although they are special in the sense that the Java layer does not have explicit classes for these types. Rather,
the implementations are entirely located in the mirror packages in the ART runtime, as shown in Figure 4.3. Arrays, whether byte or char, have similar structures as the String mirror class, as both are subclassed from the C++ mirror::Object class. We found that we cannot easily add one bit of taint tag in a member variable (e.g., count) as we did in Strings, because a significant portion of assembly code accesses variables like this. So we added one additional integer variable in mirror::Array as the taint tag, which increases every array by 32 bits due to the additional member variable. We also needed to handle a side effect: the assembly code assumes specific sizes of arrays. We solved this problem by modifying the size of arrays by re-defining the ARRAY_SIZE macro in a header file, which can be easily done in AOSP. This avoids having to further modify C++ or assembly code.
4.4.4 Propagating taints

Next, we describe how FlowPass propagates taints across function calls. We have identified five classes of functions that operate on passwords: copy, append, sub, reversible, and irreversible, and we apply different propagation logic as shown in Table 4.1. We note that a sixth class of functions, which check password strengths, are excluded from our tracking logic. FlowPass copies the taint tag of the input value to the output value for functions in the copy class, e.g., `System.arraycopy()` (Rule 1). FlowPass also taints the output if the function appends to a tainted input, or concatenates several inputs where at least one is tainted, e.g., in the case of the + operation on Strings (Rule 2). A third class of functions copy parts of the input data, e.g., `String.substring()`, and FlowPass propagates taints for these functions, too. This favors a more conservative analysis, which may result in false positives but not false negatives; we rely on the provenance tracing engine (discussed later) to identify and resolve false positives. This design decision is also adopted by Taint-Droid and TaintART (Rule 3). Reversible functions include Base64 encoding or lossless compression, for which FlowPass propagates the taint of the inputs (Rule 4). Irreversible functions, such as encryption or hashing, would disable further tracking in FlowPass (Rule 5).

Achieving a precise analysis would require us to identify each class of functions accurately. We found that, in practice, the number of APIs that we need to instrument in the Android framework is very limited, because most functions internally rely on the same set of lower-level functions. For instance, multiple copy functions (e.g., `Array.copyOf()`) eventually invoke `System.arraycopy()`. For these cases, we identify the key functions that are shared at the lowest level and instrument these. Figure 4.2 shows a complete list of our instrumented functions modules.

As a concrete example, we have shown a simplified version of a copy function in Ta-
Table 4.1: Taint propagation logic. Variables $a$ and $b$ are strings or arrays. Variable $c$ can be any type; but if it is not a string or array, it does not have a taint tag, i.e., $\tau(c) = \emptyset$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Function Type</th>
<th>Taint propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1</td>
<td>$a \leftarrow \text{copy}(b)$</td>
<td>$\tau(a) \leftarrow \tau(b)$</td>
</tr>
<tr>
<td>Rule 2</td>
<td>$a \leftarrow \text{append}(b,c)$</td>
<td>$\tau(a) \leftarrow \tau(b) \cup \tau(c)$</td>
</tr>
<tr>
<td>Rule 3</td>
<td>$a \leftarrow \text{sub}(b)$</td>
<td>$\tau(a) \leftarrow \tau(b)$</td>
</tr>
<tr>
<td>Rule 4</td>
<td>$a \leftarrow \text{reversible}(b)$</td>
<td>$\tau(a) \leftarrow \tau(b)$</td>
</tr>
<tr>
<td>Rule 5</td>
<td>$a \leftarrow \text{irreversible}(b)$</td>
<td>$\tau(a) \leftarrow \emptyset$</td>
</tr>
</tbody>
</table>

Table 4.2. Our instrumentation consists of one additional line of code that invokes the Taint-Manager, which sets or clears taint bits in the C++ and Java layers. Such instrumentation can be done very easily once we have identified the set of functions to instrument.

```java
byte [] copy(byte [] in)
{
    byte [] out = new byte [ in.length ];
    for (int i = 0; i < in.length; i++) out[i] = in[i];
    // Instrumented statement
    TaintManager.propagate(in, out);
    return out;
}
```

Table 4.2: An example instrumentation for taint support.
4.4.5 Sources and sinks

The taint sources are the user passwords and the sinks are several types of APIs that represent password leakage. Figure 4.2 shows the sources and sinks.

**Sources.** FlowPass labels only one taint source that accepts user passwords: the **EditText** class, which is the default password UI widget in Android. The passwords will then be stored in the **SpannableStringBuilder** class, and they may be later extracted as arrays or strings by invoking **getChars()** or **toString()**. Since the **EditText** class can be used for normal text entry and password entry, we distinguish between these two cases and only taint the latter usage.

**Sinks.** FlowPass has three types of taint sinks that represent password leakage:

- Storage APIs: writing data to permanent storage
- Network APIs: sending data to the network
- Logging APIs: logging data to Logcat

For storage APIs, we have annotated a low-level write function **IoBridge.write()**, because we found that all write operations in standard Java eventually call this internal function. This function is then in charge of passing the data to the native function to invoke system calls. In addition, we have also instrumented write operations to Sqlite databases.

For network APIs, we have instrumented send operations in the TCP layer and also in the TLS layer. For TLS, Android uses an external library called Conscrypt; we have added our instrumentations there. For the logging APIs, we have instrumented Logcat functions. Our modifications are simply to add one line of code that invokes the TaintManager, which does the rest of the tracking.

One limitation that FlowPass shares with TaintDroid and TaintART is that the sinks are all in the Java layer. If an app uses its own native library, e.g., a third-party SSL
library, to directly call native methods, FlowPass cannot identify the leak. Supporting this would require instrumenting the system calls for taint tracking. For the purpose of studying common misuse patterns, we have decided to support the common, Java-layer sinks only.

4.4.6 Provenance tracing

The above taint tracking components allow FlowPass to detect potential leaks. Next, we describe the provenance engine in FlowPass, which outputs an informative provenance trace of the leak, so that developers can confirm, understand, and potentially fix the leak. This provenance engine is made possible by the design decision to track taints at the function call level, avoiding the semantics gap problem in a more lower-level tracking.

FlowPass supports provenance tracking by making one more modification in the ART runtime. We have instrumented the DoCallCommon function in ART runtime, which is invoked every time a Java method is called. Then, it checks whether the callee method
contains compiled native code or only has Android bytecode. For native code, this function passes the control to the JIT engine, jumping to the native address of the callee; and for bytecode, it launches the interpreter for execution. Our provenance engine hooks this function to collect traces. If any input arguments to the callee is tainted, the provenance engine adds the callee function names and argument addresses to the TaintManager. One limitation here is that, if functions are inlined for performance, the provenance engine cannot capture them. That said, inlined functions also tend to be very simple; omitting them from the provenance might miss some functions in the traces but would increase performance.

As a concrete example, consider the provenance trace in a program that uses Volley for network I/O, shown in Figure 4.4. The provenance trace is a directed graph, where rectangle nodes are the passwords and their virtual addresses, ellipse nodes are the function calls, and directed edges are the data flows. In this example, the password node with address 0x1234 is created when the password is extracted from the widget using `toString`; the password node with address 0x4568 is created when the original password is appended with a user name using the + operation. The second password node then leaks data to the network.

4.4.7 The semi-automated workflow

Next, we describe how an analyst could drive FlowPass for semi-automated analysis. Our framework consists of a) a static analyzer, which identifies whether an app uses password-based authentication, b) a provenance visualizer that displays the tracing information, and c) as an experimental feature, a dynamic API hooking engine that can add support for analyzing third-party APIs. FlowPass is meant to be used in this framework for end-to-end analysis. Figure 4.5 shows this workflow.

**Static analyzer.** As the first step, the analyst invokes the static analyzer to filter
apps based on whether they perform password-based login. The analyzer achieves this by searching for password widgets declared in the layout files, and by checking the existence of code patterns that generate password widgets dynamically at runtime. This step runs as part of the ApkTool framework.

**Visualizer.** The app is then analyzed on FlowPass dynamically, generating provenance tracing. The dynamic analysis will output whether the app misuses passwords, and the visualizer will display the trace of the data flow. The only job of the analyst is to drive the UI manually for password login. As future work, we plan to investigate whether we can use UI automation techniques [120, 121, 122, 123] to further help the analyst in this step. Finally, the analyst could use the provenance trace to understand and fix the problem.

**Dynamic API hooking.** One limitation of FlowPass is that it loses taint information in third-party APIs that might manipulate and leak passwords. For example, if a library uses a custom URL encoding that copies the password byte by byte for URL construction, then
FlowPass would miss this. We allow the analyst to dynamically hook such APIs for further tracing, and also supply a list of APIs that we have found to leak passwords in reversible forms. This experimental feature is meant to be a proof of concept that demonstrates how one could further enhance FlowPass to cover more libraries.

4.5 Evaluation

Next, we evaluate FlowPass using a set of real-world Android apps. Our experiments are designed to answer two high-level research questions:

- How well does FlowPass meet our key design goals?
- To what extent do real-world Android apps misuse passwords?

4.5.1 Methodology

Datasets. In order to answer these questions, we have collected top Android apps from two main sources. First, we have downloaded the top-20 apps per category from the Google Play app store, totaling 560 apps. We have then downloaded 1,343 apps from two popular app stores in China: Tencent My App and MIUI app store. In total, we have downloaded 1,903 apps. We have chosen to analyze Chinese apps because the Chinese app market is the largest in the world∗, and they have a unique ecosystem because Google Play store is not present in China. Analyzing apps from these two sources would therefore allow us to see if there are different practices in the two separate app communities.

Preprocessing. We then performed a preprocessing to filter out apps that do not use password-based login and those that do not run on x86 emulators. After preprocessing, we have 103 out of 506 U.S. apps (20.3%) and 79 out of 1,343 Chinese apps (5.9%) for further

analysis. We note that the percentage is notably lower for Chinese apps because, as we discussed before, they are released only with ARM support. U.S. apps, on the other hand, are usually released with support for many architectures, such as x86, x86_64, MIPS, and ARM. The filtered apps from Google Play either do not use password-based login or can only run on Google-licensed devices with Google Play APIs.

**Workflow.** Our semi-automated framework takes a set of .apk files as input, and installs and launches the apps one by one for analysis. After launching an app, the analyst needs to manually perform the login. The framework then takes over the rest of the analysis for taint tracking and provenance tracing. Manually performing login is not difficult, but this is complicated by the fact that we cannot obtain real credentials for all tested apps. For instance, the Chase Bank and Bank of America apps require one to personally open an account to generate real credentials; some Chinese apps require working phone numbers in China to create accounts. We have sidestepped this by asking the analyst to provide fake credentials. FlowPass can taint these credentials and analyze leaks regardless. But this does cause one limitation: the login will not succeed. Therefore, FlowPass only analyzes password misuse before the login and cannot drive the app far enough to study, e.g., password retention behaviors [85]. Nevertheless, FlowPass has found interesting misuse patterns in many apps despite this limitation.

### 4.5.2 How effective is FlowPass?

Our first design goal is for FlowPass to be effective, i.e., it should be able to handle complex, real-world apps. The highlight here is that FlowPass successfully finishes its analysis on all tested apps, and it has identified common password misuse patterns, and found 13 previously unknown security bugs in popular and complex apps.

**Overheads.** First, we found that FlowPass incurs negligible overheads. Compared to
some analysis tools, which incur several times of overhead [94, 95], the performance of FlowPass is not perceptible to the analyst because it only incurs a few more function calls. In terms of memory overhead, we have measured extra memory usages and found them to be on the order of a few kilobytes, which are also negligible. These low overheads come from our design decision to keep the taint tracking lightweight and non-intrusive.

**Precision.** FlowPass reported 7 false alarms in 182 apps, achieving a false positive rate of 3.8%. We have further analyzed these 7 apps using the provenance traces and identified the root cause. Some network libraries, such as Apache, buffer some of its request to a temporary file on disk. It happens that this buffer was tainted by the passwords, although the part of the buffer that is written to the disk did not contain the actual passwords. This limitation stems from the fact that FlowPass manages one taint bit per array to achieve low overheads. Overall, it seems that the false positive rate is low enough for the analyst to inspect the provenance trace and identify these cases.

**Coverage.** In terms of coverage, we have found that FlowPass cannot completely identify all password uses in 39 apps (21.4%). Of these, 18 apps (9.9%) send the tainted data to native methods, which tools like FlowPass, TaintDroid, and TaintART cannot track because they operate at the Java layer. Also, 21 apps (11.5%) use third-party APIs that manipulate passwords byte by byte. The first category of apps can be further handled by, e.g., adding sinks at lower layers such as system calls. The second category can be handled by dynamic hooking these libraries to increase coverage. We consider these extensions as future work.

### 4.5.3 Are the provenance traces informative?

As discussed above, we have relied on the provenance traces to identify false positives in 7 apps. The provenance trace highlights parts of the graph where FlowPass is not able to trace due to native function calls. This diagnostic information allows the analyst to zoom
As a concrete case study, Figure 4.6 shows the provenance trace for the Dark Horse app, which has been installed for more than one million times. FlowPass detected a password leak and generated the provenance trace of the data flow. The password (0x12e379e8) is first used as an argument in creating an AuthenticationTask object. Then, it is appended to other objects and converted to a byte array (0x12e86230). Next, the app encodes the object with Base64 encoding, creating another tainted byte array (0x12e862b0). It then goes through a series of function calls for authentication, and is converted to a string object (0x12e87000) and written to Logcat. The yellow node with double circles represents the sink at which the password is found. As we can see, this trace is very helpful for the analyst to quickly understand how the leak happens and suggest fixes. We have reported this vulnerability based on this trace. The app vendor released the new version two days after our reporting.

4.5.4 How extensive are the instrumentations?

A third goal that FlowPass is designed for is portability. We would like our tool to be relatively easily portable across runtimes and platforms, and our approach is to concentrate the modifications to key Java objects, such as Strings, byte arrays, and char arrays. These are part of the core Java library and relatively stable over time. However, we note on two places where FlowPass has modified the ART runtime. First, in terms of Java strings, Android uses C++ code in ART to manage and optimize its strings; this is in contrast to the HotSpot JVM that manages strings with regular Java classes. Therefore, we needed to instrument the ART runtime by adding one bit in the mirror::String and mirror::Array classes. Second, to collect provenance traces, we have instrumented one function in the ART runtime to monitor all Java function calls. We have added about 600 lines of code
Figure 4.6: A provenance trace that shows how password leaks in a popular app, Dark Horse.
in C++ to the ART Runtime. Of these, the TaintManager includes 400 lines of codes for maintaining provenance data, but it is platform-independent. Therefore, 200 lines of codes are dependent in the system. Looking at previous code-base from Android 4.4 to 8.1, we found that both types are stable parts, although a clear risk to our approach is that a future Android version, looking for additional performance on method calls or with string handling, might reengineer the features that we rely upon.

4.5.5 Finding 1: Leaking passwords to the network

Next, we turn to report the findings we have obtained, applying FlowPass to the downloaded apps. Table 4.3 summarizes the high-level statistics.

<table>
<thead>
<tr>
<th>Details</th>
<th>Google Play</th>
<th>Chinese markets</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network leak (bug)</td>
<td>1 (0.9%)</td>
<td>5 (6.3%)</td>
<td>6 (3.3%)</td>
</tr>
<tr>
<td>Logcat leak (bug)</td>
<td>5 (4.9%)</td>
<td>3 (3.8%)</td>
<td>8 (4.4%)</td>
</tr>
<tr>
<td>SSL leak (bad)</td>
<td>58 (56.3%)</td>
<td>23 (29.1%)</td>
<td>81 (44.5%)</td>
</tr>
<tr>
<td>Encryption (good)</td>
<td>12 (11.6%)</td>
<td>38 (48.1%)</td>
<td>21 (21.4%)</td>
</tr>
</tbody>
</table>

Table 4.3: High-level statistics.

The first class of problems that FlowPass detected is apps leaking plaintext or reversible forms of passwords to the network. We have found six apps (3.3%) with this vulnerability. Five out of the six are Chinese apps and the remaining one is a Google Play app. However, the Google Play app, Infinite Design, is a popular art app that has been installed for more than five million times.

It is very concerning to see such popular apps fall short in following basic security practices, leaking passwords to the network without any protection. Such apps will be
vulnerable to an adversary that can sniff network traffic, e.g., one that eavesdrops over public WiFi. We have performed an additional experiment, where we analyzed another app released by the same developers, and found that this second app does send passwords via HTTPS. This seems to suggest that these developers are aware of password security practices, and that the problem in Infinite Design might be a one-time accident.

This example shows the effectiveness of our tool in detecting and analyzing the problem. Problems like this are easy to identify during the development process, and could be fixed relatively easily. However, without a systematic tool to check for these patterns, there may be accidental errors like the one we have found, which may lead to significant vulnerabilities. We have reported this issue to the company, and they have confirmed the problem and promised to fix it soon.

4.5.6 Finding 2: Logging passwords via Logcat

We found a second class of problems where apps log passwords via the Logcat utility. Logging sensitive data via Logcat is problematic, because Logcat is not private; many apps have the necessary permission to access Logcat data. In older versions of Android, Logcat data is accessible by any app!

FlowPass found that eight apps (4.4%) log plain or reversible forms of passwords via Logcat. Five of these apps are U.S. apps, whereas three are Chinese apps; four out of five U.S. apps have been installed for more than one million times. These unfortunately include the first-ranking app in the “Food” category, DoorDash, which has been installed more than ten million times. Two out of the three Chinese apps log passwords via Logcat and send passwords to the network.

Using the provenance trace, we have identified a common mistake pattern. We see that passwords first pass through the OkHttp network library, and then reach the Android
logging APIs. This suggests that OkHttp has been logging the passwords. Figure 4.7 shows the dump of the Logcat data from DoorDash with ‘OkHttp’ tags. This problem happens due to the interaction of two bad coding practices. First, the app does not disable debug messages of the network library. Second, they directly send plaintext passwords to the TLS library, hoping that they would be encrypted in that layer. This has resulted in plaintext passwords (and potentially other types of sensitive data) to be logged to Logcat data. We have further tested older releases of DoorDash and found the same vulnerability. We found that four out of the eight apps with Logcat leakage expose the password due to this cause. The other half, however, actively log the passwords themselves. FlowPass has also found one example where the password is logged in Base64 encoding.

Such results are even more alarming than they might appear, because our samples are biased towards the top-ranking popular apps. Even among these apps, 7% of them leak passwords to the network or via Logcat. It seems safe to assume that the less popular apps, where the developers presumably might be less skillful, could even have more leaks. A tool like FlowPass would come in handy for these developers to check their password uses in the development cycle.

4.5.7 Finding 3: Sending passwords via TLS

A third class of misuse is to send plaintext passwords to the SSL/TLS library. This is not as serious a leak as the problems above, because the passwords will not enter the network un-
encrypted. However, solely relying on TLS for encryption is not sufficient, because TLS is vulnerable to many attacks. At a simple example, we have already seen how Logcat might log passwords in plaintext before they enter the TLS library. Overall, 81 apps (44.5%) fall into this category. As further validation, we have emulated an attacker that can compromise TLS by hooking the TLS library and capturing packets sent to TLS. We printed all packets in this channel and observed plaintext passwords in all these apps. Figure 4.8 shows the captured packet during the login with the Chase banking app. As a secure practice, an app should derive secrets from passwords for authentication instead of sending passwords directly.

4.5.8 Further observations

We have further broken down the results per category for Google Play apps and in the Chinese app markets. Figure 4.9a and 4.9b show the results. There are two high-level observations.

**Observation #1:** *Chinese developers tend to use stronger protocols for login.* In terms of leaking passwords via the network or Logcat, this vulnerability exists in U.S. and

```plaintext
POST /auth/fcc/login HTTP/1.1.. x-jpmc-csrf-token: AntiCsrfToken.. Accept: */*..Cache-Control: no-cache..User-Agent: ua=Dalvik/2.1.0(Linux; U; Android 9; AOSP on IA Emulator Build/PQ2A.190305.002);av=3 80000051;dv=Android 9;did=a353d791dfe6d2b5;oid=07-25-2019..Accept-Encoding: gzip.....Referer=https%3A%2F%2Fwww.chase.com&auth_otp_reason=GATEWAYAuth_mobile_mis=DEVMAKE%4DAOS+on+IA+Emulator&auth_passwd=OurFakePassword&deviceTimeStamp=201907250000%3A55&deviceAppVersion=...&auth_userId=OurFakeUsername&authdeviceSignature=%78+%22navigator%22%3A+%713%7D%2C%22plugins%2
```

Figure 4.8: Experiment with Chase bank.
Figure 4.9: Comparisons between Google Play and Chinese apps. Since the app stores have different categories, we have merged similar categories for easy comparison. Red bars are vulnerable apps that directly leak passwords to the network or via Logcat. Yellow bars are apps that send passwords to TLS. Blue bars are the apps that encrypt passwords. The remaining grey bars are the apps that our framework did not fully cover.
Chinese apps in roughly similar percentages. However, a significantly higher percentage (48.1%) of Chinese apps additionally encrypt passwords before invoking TLS. In comparison, only (11.6%) of the U.S. app perform encryption. A notable percentage (56.3%) of U.S. apps only rely on TLS. Although it is hard to draw statistically significant conclusions about all apps, the analyzed apps seem to suggest that Chinese app developers have a higher awareness of secure coding practices, at least in terms of whether to rely only on TLS.

**Observation #2:** Financial apps are not safer than others. Another worrisome result is about financial apps in the U.S. app market. Security is much more critical for financial apps, as a password compromise would directly result in economic loss. Therefore, before the analysis, we expected that these apps would take more effort and implement secure login implementation such as SRP, or at least some form of challenge-response schemes. However, 10 out of 13 apps, including Chase, BoA, Schwab, PayPal, and Venmo, directly send user passwords via TLS.

### 4.5.9 Bug reporting

We have reported the vulnerabilities to the six U.S. app vendors, and most of them have fixed their problem soon. One vendor has additionally provided us with a bug bounty as reward. This shows that vendors are aware of the significance of the problems, and are willing to fix the problems or eliminate them before release. However, developers can make mistakes for various reasons, and a tool like FlowPass would be a convenient way for developers to check their apps before release. We are in the process of disclosing the vulnerabilities to the Chinese app vendors, and will include more discussion here.
4.5.10 Summary

To summarize, there are two key takeaways from our evaluation of our tool on popular apps.

First, our lightweight taint tracking approach is effective for identifying password misuse in real-world apps. The provenance traces can further help developers to understand and fix the problems. Limitations exist in terms of false positives and negatives, but the provenance traces provide useful diagnostic clues to identify these cases.

Second, our analysis reveals many bad practices in top-ranking apps. Developers make critical mistakes, and they lead to password leakage. This calls for a solution that can help developers identify and analyze misuses before release, and as a further step, it would be helpful to have built-in support in the Android framework that encourage secure password uses.

4.6 Discussion

Misusing passwords after login. Certainly, apps may misuse passwords after they have finished the authentication step. Since an analyst using FlowPass typically creates fake credentials for testing, FlowPass cannot automatically drive an app past the login phase to determine if there is subsequent undesirable behavior that only happens once properly logged in. Certainly, an app’s developer will have access to real accounts for testing, but a third-party analyst might not have a real account. Nevertheless, we have shown that FlowPass is powerful enough to shed light on common misuse patterns even for the login phase.

Larger-scale analysis. We have obtained insightful results using FlowPass on hundreds of apps. We believe that there may be even more interesting findings to scale the analysis
up to millions of apps. Our current system requires a few minutes of human effort per app, which is fine at our current scale, but would be prohibitive if we wished to analyze every Android app. To achieve this scale, the main difficulty lies in the need for automating the login process, which is often guarded by anti-robot features, CAPTCHAs, and the like. Such automation may be possible using accessibility features meant to make apps usable by users with disabilities, possibly combined with random input generators. Alternatively, it would be interesting to see how effective a lightweight static analysis would be for such tasks, allowing us to identify password-relevant parts of the control flow graph, and then to perhaps use this in combination with a runtime approach to identify inputs that result in passwords being processed.

Prototype limitation. The current FlowPass prototype works for an x86 emulator, but not for ARM. This has posed a limitation in analyzing many Chinese apps that only support ARM. However, modifying FlowPass to support ARM should not be difficult. We expect that the engineering challenge would mainly come from the ARM emulator codebase in AOSP, which is not very mature and runs slowly, or would require porting FlowPass to run directly on Android phone hardware.

4.7 Related Work

There are several threads of work related to FlowPass.

4.7.1 Dynamic analysis

Dynamic analysis of Android apps has been studied in DroidScope [95], TaintDroid [70], TaintART [93], Artist [113], Malton [94], and in many other work that extend TaintDroid [46, 71, 122, 124, 125, 126, 127]. In particular, Spandex [71] focuses on password analysis as well. Other work develop dynamic analysis engines that do not modify the Android
system but rather the Android apps [128, 129]. FlowPass, in contrast to these work, has a unique angle in its use of lightweight taint tracking to analyze password misuse, achieving an efficient and portable solution.

4.7.2 Static analysis

Another line of work uses static analysis of the apps to identify information flow, such as AAPL [130], AmanDroid [88], ComDroid [68], DroidSafe [89], EdgeMiner [131], FlowDroid [67], IccTA [132], and Horndroid [133]. Static analysis has also been used to detect specific malicious behaviors or misuse. For instance, CHEX [134] detects component hijacking vulnerabilities, Stowaway [135] analyzes API calls to detect overprivileged applications. MalloDroid [6] detects misuse of TLS APIs, CryptLint [9] detects misuse of cryptographic APIs, Bianchi et al. [11] detect misuse of fingerprint APIs, and LeakScope [86] detects misuse of cloud APIs. Static analysis cannot easily handle obfuscation and dynamic loading. Also, these work assume that the analyst already knows specific patterns of bad behaviors. A tool like FlowPass could offer help in identifying these patterns.

4.7.3 Hybrid approach

Existing work like Drebin [69], DroidMiner [136], RiskRanker [137], Zhou et al. [138] combine static analysis with machine learning, or heuristics (e.g., the presence of crypto methods) to detect malicious Android apps. AppAudit [139] combines static and dynamic analyses to avoid over-estimation. Similarly, SMV-Hunter [140] uses static analysis to detect bugs and dynamic analysis to confirm the vulnerability. TriggerScope [141] detects hidden “logic bombs” in sophisticated malware by combining static analysis with symbolic execution. AppsPlayground [122] and Mobile-sandbox [124] address scalability concerns by combining TaintDroid with fuzzing. PUMA [120], Sapienz [121], Stoat [123] provide
advance test automation for Android beyond random exploration. We believe that it is an interesting direction to integrate FlowPass with static analysis as future work.

4.7.4 Security flaws in applications

A range of work has looked at specific types of security flaws in Android apps. Reaves et al. [80] analyze information leakage in banking apps. Fahl et al. [82] reveal that many password manager apps are vulnerable to clipboard sniffing attacks. Silver et al. [83] has found critical flaws in the auto-fill functionality in password manager apps. Recent usability studies have also looked at why developers make mistakes, by analyzing the patterns of misuse [61, 72, 76, 81]. To prevent developers mistake, researchers also proposed an IDE extension [87] to help developers follow good security practices.

4.8 Conclusion

Password-based authentication is a prevalent practice in Android. User passwords are sensitive data, and their compromise may have severe consequences. We have designed FlowPass, which can detect and analyze password misuse in Android apps. FlowPass performs lightweight taint tracking by enhancing key data types where passwords typically reside. It propagates taints at the function call level to analyze password usage patterns, and generates provenance traces to offer diagnostic information for the analyst. We have applied FlowPass to 182 Android apps, and identified common patterns of password misuse. FlowPass found 13 previously unknown bugs in popular apps that have been installed for more than one million times. We have reported these vulnerabilities to the app vendors, and most of them have fixed the problems shortly afterwards.
Chapter 5

Conclusion

In this thesis, I present my studies with various Android apps and the Android framework to understand the misuse of secrets in the mobile environment. My research found that the Android operating system does not provide enough security features in protecting sensitive data. Also, this dissertation confirms that developers repeat mistakes in managing sensitive data, exposing cryptographic secrets, user passwords. Even security experts and developers of popular apps make mistakes.

To address these problems, this dissertation suggests two types of practical solutions: by abstracting and by detecting. Firstly, sensitive data can be protected by abstracting critical routines in the operating system level and providing a set of APIs to developers. We present a prototype design called KeyExporter, which is an abstraction layer for user password management. The evaluation result shows that it successfully eliminates password retention problems in vulnerable apps with minor modifications. Secondly, it is very important to detect developers’ mistakes at the early stage of the development cycle. For that, we have designed FlowPass, which is a practical and informative dynamic taint analysis tool. FlowPass shows how a systematic approach can enhance user privacy by finding previously unknown bugs in complex and popular apps.
Bibliography


on System Sciences, 2017.


