# Total Recall: Persistence of Passwords in Android

Jaeho Lee Ang Chen Dan S. Wallach
Rice University
{jaeho.lee, angchen, dwallach}@rice.edu

***Abstract*—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. This paper looks at unnecessary password retention in Android phones by popular apps, secure password management apps, and even the lockscreen system process. We 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. We have developed solutions that successfully fix these problems with modest code changes.**

## I. 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 [5], Meltdown [38], and Spectre [33]. If adversaries can gain physical access to a device, they may also be able to dump its memory directly, e.g., via a "cold boot" attack [28], [41], or even through its USB connection [29]. 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., [30]).

Therefore, we should delete sensitive data 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 [44] and GnuTLS [39], explicitly zero out key material after a session ends. In a garbage-collected system, these issues are even more serious, as old copies might be left behind in memory. Aware of this issue, the Java Cryptography Architecture (JCA) [45], in 2003, was engineered to use mutable character arrays rather than `String` objects, which are immutable, for the explicit purpose of making its keys easier to overwrite.

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., [9]), password-based authentication is still the de facto practice for many applications. In addition to the direct use and transmission of plaintext passwords, applications will also use passwords for "key stretching" [32] (see, e.g., PBKDF2 [31], bcrypt [50], and scrypt [46]), ensuring that a captured password does not also allow for the decryption of prior recorded sessions.

Cryptographic libraries have integrated many well-understood security practices (e.g., constant-time cryptography [48]), and developers tend to stick to relatively mature libraries (e.g., OpenSSL). When it comes to password-based authentication, developers may be tempted to follow idiosyncratic security practices, unaware of the dangers of keeping passwords live in memory. Given that app developers have different levels of experience, and that there is a large number of apps in the market, security of authentication features can be expected to vary considerably across applications. Indeed, recent studies have repeatedly found that developers implement security features poorly. For example, apps have been reported to misuse TLS libraries [21], [22], [25], cryptographic APIs [19], OAuth protocols [11], and fingerprint APIs [8]. A recent study has also revealed that some developers simply store passwords in plaintext on disk [43]. In this paper, we ask a question: *How well does Android manage passwords?*

We perform a systematic study on how the Android platform and popular Android apps handle user passwords. The Android platform has many complex layers that interact (e.g., the Dalvik/ART runtime system, the operating system kernel, and the applications), so poor practices in any of these layers could lead to security issues. Furthermore, Android apps have a complex lifecycle; an app might be put into "background" or even "stopped" without necessarily being given an opportunity to clean up sensitive values in its memory prior to the lifecycle change. 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 for longer than necessary. Last but not least, previous studies have found that Android apps fall short in performing secure deallocation [57], and that they may retain TLS key materials in memory [34].

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 [16], [24], [54], so a stolen password would cause widespread damage. We have also identified the common root causes, such as the insufficient security support in Android widgets and the widespread use of `String`

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23180
www.ndss-symposium.org

objects to store passwords. We propose solutions that fix the Android framework and the studied apps with modest code changes. We also present a design that we call KeyExporter, which manages passwords securely and can be integrated with vulnerable apps to eliminate password retention. KeyExporter integrates cryptographic primitives with the password widget and exports key materials using password-based key derivation functions (e.g., PBKDF2, scrypt) and password-authenticated key agreement (e.g., SRP [61]). Our evaluation shows that our solution eliminates password retention in all of the apps that we tested, hardening the system against memory disclosure attacks.

Concretely, we make the following contributions in this paper, after describing more background in Section II.

- A demonstration of password retention problem by analyzing the memory dumps of 11 popular apps (Section III);
- A comprehensive analysis of the Android framework and a variety of Android apps; the identification of common root causes (Sections IV+V);
- Our solutions: SecureTextView, a secure version of Android widgets that can eliminate password retention, and KeyExporter, which can remove passwords in Android apps (Section VI);
- Implementation and evaluation of our solutions, which successfully achieves the goal of timely password deletion in all tested apps (Section VII);

We then provide a discussion in Section VIII and describe related work in Section IX, before concluding in Section X.

## II. Background and Motivation

In this section, we present more background on Android authentication, and discuss how passwords may be retained by each stage of the Android app lifecycle.

### A. Authentication in Android

Recent Android versions have started the use of fingerprints, face recognition, and voice recognition as means of authentication. However, to date, *passwords* are still the mainstay for Android authentication, thus our main focus in this paper. Broadly, Android authentication apps fall 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 1 shows a typical workflow of remote authentication, which has three main stages. ① The app prompts the user to enter their 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 their password again until the shared temporary key expires.

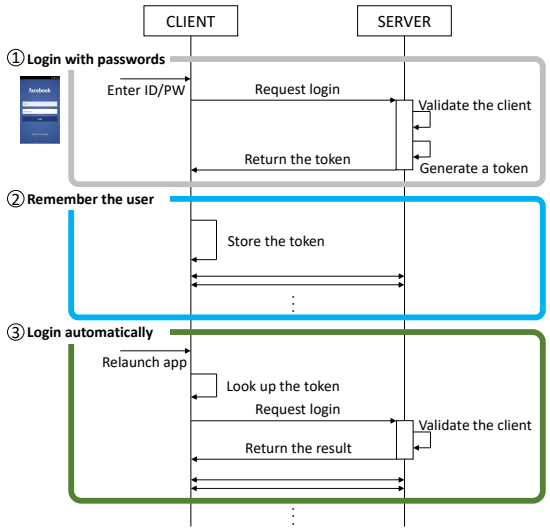



Fig. 1: Authentication steps in client applications.

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 one of many forms of two-factor authentication. Others will construct ad-hoc challenge/response protocols with hashes or other cryptographic primitives. Better apps will 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's identity (e.g., Facebook or Google). Regardless of these details, any app needs to ensure that user passwords are deleted properly and promptly, despite all of these complexities.

**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 [32]. Such apps can also help users generate random passwords to mitigate password reuse [16]. As concrete examples, two popular apps in this category are 1Password[1] and LastPass[2], 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 become vulnerable.

[1]See https://support.1password.com/pbkdf2/
[2]See https://lastpass.com/support.php?cmd=showfaq&id=6926

### B. Risks of Password Retention

Unfortunately, there are many opportunities for passwords to be retained by Android for longer than necessary.

**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 hidden and it stops running, but the underlying process and its memory are still present. When it is resumed, it again has the chance to run, draw to the screen, and so forth. If an app is still holding user passwords when it is "paused", the passwords may remain live in memory for an extended period of time. Although Android may destroy certain background processes, it typically does so only if 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, regular Android apps do not use TrustZone to manage their passwords.

**Delayed garbage collection.** Most Android apps are written in Java, so their memory is managed by a garbage collector (GC). Therefore, even when an app deletes its last reference to a password, the memory will remain uncleared until the GC reuses it. This delay can last for minutes or even hours, depending on the memory pressure of the GC system. Furthermore, in the (seemingly intuitive) case of using Java's `String` class to hold passwords, developers cannot manually overwrite them, because `String` objects are immutable. Thus, the Java Cryptographic Architecture [45] recommends that passwords should be stored in `char` arrays instead of `String` objects. However, 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 rather than strings, they will find less support for this style of coding from standard libraries.

**Java vs. native code.** Although Android apps are commonly written in Java, they may make native calls to underlying C libraries included by 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 [34].

**Long chain of custody.** User inputs may also be unintentionally buffered by various processes and retained in memory. For instance, when a user inputs their 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. If any of the processes in this chain of custody accidentally retains the password, it may persist in memory.

## III. Password Retention on Android

Next, we describe our threat model, and demonstrate that password retention is a widespread problem in Android.

### A. Threat model

We assume that an adversary can perform memory disclosure attacks on an Android device, e.g., by exploiting software vulnerabilities [5], launching side-channel attacks [33], [38], or after the physical capture of a device [28], [41]. Plenty of evidence suggests that such attacks are feasible on Android. For instance, the recent memory dumping vulnerability in the Nexus 5X phone [29] allows an attacker to obtain the full memory dump of the device even if the phone is locked. As another example, vulnerability in WiFi chipsets [5] can allow attackers to obtain a memory snapshot of a victim device remotely. An attacker can then analyze the memory snapshot and obtain any sensitive data left uncleaned, such as user passwords.

### B. Initial Memory Dump Analysis

We selected 11 Android apps for a preliminary study of password retention problem. Six of the apps are very popular—having more than 10 million installations each—and other four apps are password managers that store highly sensitive user data. In addition to these apps, we tested the system processes that are in charge of unlocking the phone after receiving the correct password, which are critical to the overall security of the device. 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 installed and launched each app, and manually entered passwords for authentication. After this, we performed a full physical memory dump [56] as well as a per-process dump [34]. Our simple "black-box" approach does not make any assumption about the apps. If the passwords are anywhere in memory, we will find them. 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").

### C. Results and Observations

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

**Observation #1: All tested apps are vulnerable.** With the simple technique, we successfully retrieved the cleartext passwords for all the apps. Very popular apps, such as Facebook, Chrome, and Gmail, which have been installed more than one billion times, retain login passwords in memory. Secure password managers expose master passwords which are typically used to decrypt their internal password databases, so an

attacker would be able to capture the master password and gain access to the full databases. Moreover, the lockscreen process also leaves the PIN passwords in memory. Since the PIN password is used for full-disk encryption and decryption as well as unlocking the phone, Android spends an extraordinary amount of effort to protect the PIN—e.g., the `Gatekeeper` service verifies hashes of user passwords in TrustZone to protect them. Therefore, in the presence of memory disclosure attacks, the retention of PIN passwords in memory completely defeats the purpose of the added security measures.[3]

**Observation #2: All tested Android versions are vulnerable.** Although Table I 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 paper, we only present our findings on Android 8.[4]

**Observation #3: 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 the app into the background. This suggests that the problem of password retention seems to have gained attention from at least some Android developers, and at least *can* be solved. We would prefer the Android system to provide all app developers assistance in solving these problems.

**Observation #4: 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.` i.e., a UTF16 encoding. In addition to the full password matches presented in Table I, we have also found fragments of passwords (i.e., a prefix or a suffix of the password remains in memory rather than the full password). This appears to result from the use of `SpannableStringBuilder`, which we will describe in more detail in Section IV.

**Summary.** Overall, the above findings are worrying evidence that the password retention problem is widespread in Android. Previous studies [3], [57] raised similar issues several years ago for Android. Unfortunately, our finding shows that the problem seems to have worsened today; Of note, Tang et al. [57] looked at the Facebook and Gmail apps in 2012, concluding that they had no problems, but both apps have problems today.

What causes password retention? Is password exposure inevitable? We consider these questions next.

## IV. In-Depth Analysis: Android Framework

In order to achieve a thorough understanding of the root causes of the password retention, we have performed an in-depth analysis of the Android framework and several apps. In this section, we focus on the Android framework, describing our methodology for its analysis and our findings.

[3]We observed that PIN passwords disappear after about an hour. One previous study considers a piece of data to be exposed if it persists in memory for more than ten minutes [57].

[4]Android 9 was recently finalized. We would expect similar results, but have not yet analyzed it.

| Application | Description | Installs | Login | BG | YouTube | Lock |
|---|---|---|---|---|---|---|
| Gmail | email | 1,000 M | 6 | 6 | 6 | 6 |
| Chrome | browser | 1,000 M | 4 | 3 | 3 | 3 |
| Chase Bank | finance | 10 M | 5 | 0 | 0 | 0 |
| Facebook | social | 1,000 M | 6 | 5 | 2 | 2 |
| Tumblr | social | 100 M | 4 | 1 | 1 | 1 |
| Yelp | social | 10 M | 3 | 2 | 1 | 1 |
| 1Password | password | 1 M | 4 | 1 | 1 | 1 |
| Dashlane | password | 1 M | 2 | 0 | 0 | 0 |
| Keepass2Android† | password | 1 M | 1 | 0 | 0 | 0 |
| PasswdSafe† | password | 0.1 M | 12 | 2 | 1 | 1 |
| Unlocking phone† | system | Built-in | 7 | 2 | 2 | 1 |

TABLE I: Password exposure in popular apps. The count indicates the number of copies of the password found in memory. The columns indicate increasing opportunities and pressure for the system to reuse memory. (†) indicates apps for which source code is available.

### A. Methodology

In order to identify where password retention occurs, 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 have the same length and unique contents, so we can easily locate a password fragment within the component that's using it.

An obvious question might be why we did not use an automated analysis tool, whether based on static analysis [4], [6], [12] or dynamic analysis [15], [20], [62], [63]. Ultimately, a static analysis tool can only find a code pattern that we know in advance. Similarly, dynamic approach such as taint analysis can track all uses of sensitive data, which would certainly help us follow passwords and their derivatives. However, suitable tooling that we might adopt for our experiments appears to be experimental, either targeting outdated Android versions (e.g., NDroid [62] for Android 5.0) or requiring very specific hardware (e.g., DroidScope [63] only supports an Acer 4830T).

Of course, once we understand the root coding patterns that lead to password retention, we could then imagine creating automatic tools to highlight these patterns, efficiently, across millions of apps, perhaps even built directly into Android Studio and other development tools that Android developers use. While our manual approach lacks scalability, it allows us to trace the flow of passwords through Android's various subsystems, pinpointing a variety of relevant problems. Future automated approaches can then be built based on our findings.

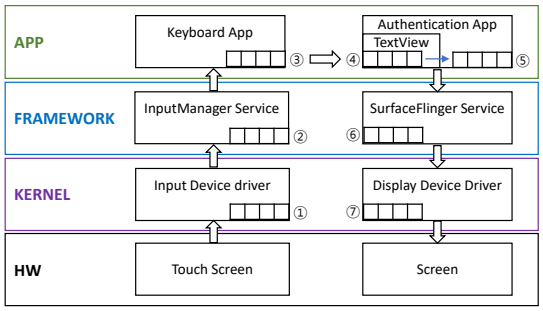



Fig. 2: The dataflow of a user password input on Android.

### B. The Android Framework

Figure 2 shows the flow of data in Android when a user types their 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 can 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 reuses the `TextView` class. This class contains about 12,000 lines of code (LoC); as it is not designed exclusively with passwords in mind, the `TextView` codebase contains many instances of insecure password handling.

For example, 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 are not given any particularly special handling, we shouldn't be surprised if there are problems. We now describe three issues in detail.

**Problem #1: 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 actually 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 the previous array without clearing it. This is the root cause of why we see fragmented passwords in memory. We also 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 mistakenly believes this method to imply secure deletion, the password will still remain in memory.

**Problem #3: Lack of secure `getPassword` API.** 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 [45] 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 the Java 1.2 release in 1998, with the suggested replacement of `getPassword()`, which returns a character array instead. Unfortunately, Android lacks such a `getPassword` API in its library, so developers might mistakenly use the *`String`* type to store passwords. We will discuss this issue further in Problem #5.

### C. Keyboard (IME) Applications

Next, we analyzed the input channel between `TextView` and keyboard apps. We found that the input channel is tightly coupled with the buffer of `TextView`; fortunately, it 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 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 passwords.

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

| Application | Company | Installs | ID | PW | Description |
|---|---|---|---|---|---|
| LatinIME | Google | N.A. | 0 | 2 | Default in AOSP |
| Gboard | Google | >1B | 1 | 0 | Default in Android 8 |
| SwiftKey | SwiftKey | 300M | 8 | 0 | |
| Go | GOMO Apps | 100M | 0 | 1 | |
| Kika | Kika AI Team | 100M | 0 | 1 | |
| TouchPal | TouchPal | 100M | 2 | 4 | |
| Cheetah | Cheetah | 50M | 2 | 7 | |
| FaceMoji | FaceMoji | 10M | 0 | 1 | |
| New Keyboard | 2018 Keyboard | 10M | 0 | 1 | |
| Simeji | Baidu | 10M | 0 | 0 | Japanese |
| Simplified Chinese | Linpus | 0.1M | **117** | **135** | Chinese |
| Baidu Voice | Baidu | 0.1M | 0 | 2 | Voice support |
| TS Korean | Team Space | 0.01M | 3 | 0 | Korean |

TABLE II: Results for the tested keyboard applications.

is the default keyboard in the Android Open Source Project (AOSP), and which should not be confused with Google's Gboard, which has over a billion installs, according to the Google Play Store. All tested apps are listed in Table II. 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 II shows the number of copies of our user account (ID) and password (PW) that we discovered for each keyboard app. Out of the 13 keyboard 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 sensitive like passwords, so their presence in memory isn't security-relevant. Our most worrisome example is a keyboard that supports simplified Chinese—it kept more than 100 copies of the user password in memory.

**Problem #4: Buffering the most recent input.** Surprisingly, the LatinIME keyboard also has password retention issues, 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, only clearing it when the keyboard returns for subsequent data entry. This means that when the keyboard is used for a sensitive input, like entering a password, the password can stay in memory for quite a long time.

LatinIME, by virtue of being Google's AOSP reference keyboard, has been used by many third-party keyboard developers as the starting point for their own efforts. That means that we might expect a large number of less popular keyboard apps to share these same problems. Since most keyboard apps are closed source, we cannot directly verify whether their developers have done this, but we believe that Table II 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's buffering strategy.

## V. In-Depth Analysis: Android Apps

Next, we analyze several third-party Android apps.

### A. Methodology

We wish to consider "example" apps using four categories of authentication techniques: 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.

If there exist popular open-source apps in the category (e.g., Keepass2Android and PasswdSafe are open-source local authentication apps), we directly analyze their source code. Otherwise, we obtain similar types of apps from official sites, open source repositories (e.g., GitHub), security guidebooks, and developers' website such as Stack Overflow. 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 [23]. Collectively, we have analyzed more than 20 apps or code snippets, and identified common mistakes throughout the categories.

### B. Basic Password-based Authentication Apps

We first consider basic authentication apps, where an app sends raw user passwords to a server via HTTP/HTTPS. The majority of apps we collected fall into this category, because directly sending passwords is the simplest (but an insecure) way of authentication. These apps use different libraries for their implementations, but they share similar authentication steps. Sample apps 1 – 2 in Table III 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 in the wild we have examined in Section III, 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 the impacts of our modifications.

**Problem #5: Use and propagation of String passwords.** We started by focusing our attention on the use of String passwords. We mentioned in Problem #3 that the Android framework provides `getText().toString()` in `TextView`. Indeed, we saw that all the sample apps stored their passwords as Java strings by invoking `getText().toString()`. How much does the usage of `String` contribute to password retention? To measure the effect of `String` usage, we deleted all uses of `String` in sample app 1, and instead, we had the app send an empty password. The "server" ran on a local desktop machine in our lab; it always sent successful authentication messages to the app. Sample app $1_a$ in the same table shows the results after the modification. This change eliminated more than half of the in-memory passwords relative to the original

| Category | Application | Description | Login | BG | YouTube | Lock |
|---|---|---|---|---|---|---|
| Basic | Sample 1 | Uses Volley lib. | 25 | 24 | 11 | 11 |
| | Sample 2 | Uses Apache lib. | 28 | 16 | 13 | 12 |
| | Sample $1_a$ | Empty request | 11 | 10 | 5 | 5 |
| | Sample $1_b$ | Nullifying widget | 4 | 4 | 0 | 0 |
| Cha./Resp. | Sample 3 | App security book | 21 | 20 | 8 | 7 |
| OAuth | Sample 4 | With Facebook | 3 | 2 | 1 | 1 |

TABLE III: Results for some of the sample apps.

version (sample app 1). This demonstrates that `String` is a major source of problems, though not the only one.

**Problem #6: Lack of manual TextView cleanup.** Recall that `TextView` holds the password in its buffer after it's no longer needed (Problem #1 and #2). This leaves the responsibility of cleaning up `TextView` to individual app developers. To see how effective a manual cleanup would be, we modified the sample app $1_a$ to call the `clear()` method in the buffer of `TextView` right after login. Sample app $1_b$, 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. This means that calling the `clear()` method of the buffer is effective. Unfortunately, almost all apps we analyzed (including the `SystemUI` core service) did not clean up the `TextView` buffer, with Keepass2Android being the only exception. We also note that calling `clear()` does not clean up passwords in app $1_b$ immediately. This is because the `clear()` method of the `TextView` buffer does not implement secure deletion, but only sets the buffer to `null` and leaves it to the garbage collector (Problem #1).

**Problem #7: Lack of app-level zeroization.** Even if an app developer uses `char` arrays instead of `String` objects, they would also need to clean up the passwords in their apps manually, e.g., by zeroing out the `char` arrays. In other words, even with a much stronger `TextView` implementation, developers may still accidentally hold passwords in memory.

### C. Challenge-Response Authentication Apps

We now turn to apps that use some form of challenge/response authentication. 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. By avoiding the transmission of passwords over the network, they certainly improve an application's security posture, but do they do a better job of handling passwords? We analyzed an app from a popular security guidebook [27] (sample app 3). As shown in Table III, 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 those 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.

### D. 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 4 using the Facebook OAuth library, 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 the user for a password and performs the requested authentication; the Facebook app redirects back to our original app upon success, which then displays a successful login message.

Sample app 4 in Table III 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 after the phone is locked. All of these passwords were found in the memory of the Facebook app itself, not in our sample app.

Since the Facebook OAuth library and app are 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 the passwords in its memory, and this issue will impact any app that uses it. As additional evidence, sample app 4 (Table III) has identical password retention patterns as Yelp (Table I), both of which use Facebook OAuth for authentication.

### E. Password Managers

We now consider password managers. If any app developers would take caution in controlling the presence of passwords in memory, certainly it would be the developers of password managers! As shown in Table I, 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 the source code. First, this app implements a manual reference

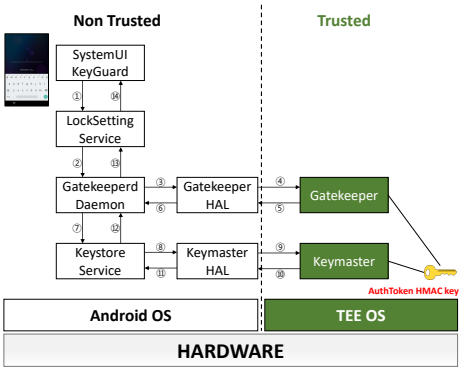



Fig. 3: Authentication steps when unlocking a device.

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.

### F. 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, 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 `LocalSettingsService` 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 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.

### G. Summary of the Problems

To summarize, our analysis has revealed seven main causes of password retention, which are prevalent in many apps and the Android framework itself. Some of the root causes are also inter-related. For instance, Problem #5 (the use of `String` passwords) needs to be addressed before Problem #7 (manual zeroization) can be solved, because `String` objects are immutable. As another example, Problem #5 is attributed partly to Problem #3 (the lack of secure API). Therefore, in order to solve the password retention, we need to develop a complete solution that addresses all of the identified root causes together.

## VI. Our Solution

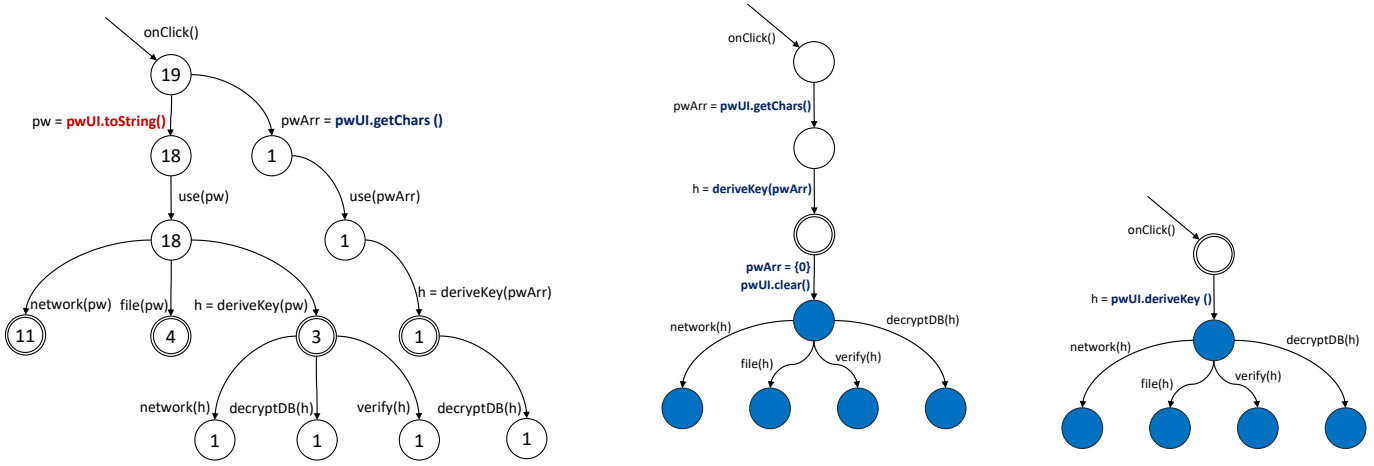
In this section, we describe a series of changes that we engineered to the AOSP framework that can address the problems we have observed.

### A. SecureTextView: Fixing the Android Framework

We have developed fixes for the Android framework by designing a patched version of `TextView` that we call SecureTextView. 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, zeroing out sensitive memory after use. This addresses Problem #1. SecureTextView 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 describe our fixes for Problem #3 in Section VI-B, which will address the use of `String` objects.

SecureTextView 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 SecureTextView 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`. However, it avoids leaving password fragments, and it contains a secure `close` method that cleans up passwords. Moreover, SecureTextView has an event handler that listens for status changes of the phone. If the phone is locked, or if an app becomes inactive, SecureTextView automatically zeroizes its buffer to clean up password fields. SecureTextView can be used as a drop-in replacement of `TextView`, as all code changes are localized to the implementation of `TextView` and `SpannableStringBuilder`. Overall, SecureTextView 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. 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




(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.

Fig. 4: The password usage patterns in the studied apps, as well as the pattern and encapsulation that KeyExporter achieves.

should not be stored in a file, but they should ideally also include suggestions on avoiding password inputs from being unnecessarily retained in memory.

### B. *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. We summarize security practices to avoid password retention:

- Using `char` array to hold passwords obtained from `TextView` (Use `charAt()` instead of `toString()`).
- Clearing the `TextView`'s buffer by calling its `clear()` function.
- Deriving a strong key (e.g., "key stretching" [32]) at the early stage without any unnecessary password propagation.
- Zeroing out all password memory.

Unfortunately, many 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 non-volatile storage. A recent survey also confirms that developers think of functionality first, and regard security as a secondary task [43]. Therefore, we believe that a general, easy-to-use solution is necessary for developers to follow the security practices to avoid password retention.

We achieve this by designing and implementing KeyExporter, with two explicit design goals: proactive security, and usability. KeyExporter proactively manages passwords internally, and only exports password-derived keys to developers; it also securely manages its 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 [26], [42]. KeyExporter is designed with the understanding that usability and simplicity can effectively promote security [1].

**KeyExporter Design.** We start by identifying common patterns of password usage in today's apps. Figure 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 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 [43]. 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" [32] 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; passwords are not made available to the application, which can only fetch cryptographic keys derived from the passwords. Figure 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 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 `getKey()` to obtain derived keys without ever accessing the raw passwords. All passwords are automatically zeroed out by KeyExporter.

**KeyExporter Implementation.** `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 [61], 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.

Figure 5 uses code segments to demonstrate how an app could use KeyExporter. Figure 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 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 5c shows another 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.

```
OnLogin:
  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:
  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:
  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.

Fig. 5: Integration with KeyExporter is easy.

## VII. Evaluation of SecureTextView and KeyExporter

We now report results from our experimental evaluation of SecureTextView 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?

### A. Methodology

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 SecureTextView (written as $A'$).
- Running a modified version of the app that is integrated with KeyExporter and SecureTextView (written as $A^{\dagger}$).

Integration of KeyExporter with an existing app requires code modification to the app. In order to integrate with KeyExporter, we needed 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, likely far more work than making the necessary client-

| Category | Application | Description | Login | BG | YouTube | Lock |
|---|---|---|---|---|---|---|
| Basic | Sample 1 | Original | 25 | 24 | 11 | 11 |
| | Sample 1′ | SecureTextView | 14 | 13 | 4 | 4 |
| | Sample 1† | STV + KeyExporter | **0** | **0** | **0** | **0** |
| Cha./Resp. | Sample 3 | Original | 21 | 20 | 8 | 7 |
| | Sample 3′ | SecureTextView | 11 | 9 | 8 | 5 |
| | Sample 3† | STV + KeyExporter | **0** | **0** | **0** | **0** |
| OAuth | Yelp | Original | 3 | 2 | 1 | 1 |
| | Yelp′ | SecureTextView | 2 | 1 | 1 | 1 |
| | Yelp† | STV + KeyExporter | N.A. | N.A. | N.A. | N.A. |
| Standalone | PasswdSafe | Original | 12 | 2 | 1 | 1 |
| | PasswdSafe′ | SecureTextView | 2 | 1 | 1 | 1 |
| | PasswdSafe† | STV + KeyExporter | **0** | **0** | **0** | **0** |
| System app | SystemUI | Original | 7 | 2 | 2 | 1 |
| | SystemUI† | STV + KeyExporter | **0** | **0** | **0** | **0** |
| PAKE | Sample $1_{srp}$ | STV + KeyExporter | **0** | **0** | **0** | **0** |

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

```
// 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.

Fig. 6: Integrating PasswdSafe with KeyExporter.

side fixes.

### B. *Effectiveness*

Table IV 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 IV, the original app contains a large number of passwords. However, the number is reduced roughly by half after using SecureTextView. 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. The entire change required less than an hour of effort.

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 3 uses a challenge/response authentication. When running app 3′, which is the version that uses SecureTextView, the number of passwords in memory also gets reduced by about half. Integrating the app with KeyExporter eliminates the rest of the passwords. After applying our fixes, app 3† is 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 again required less than an hour of effort to fix.

**#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 SecureTextView. 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 SecureTextView 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 IV show. After integrating this app with SecureTextView 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 6 shows our core changes by comparing the original code (6a) and our modified code (6b). We first located the code for the login file using a simple pattern match, and identified `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 changed only 50 lines of code, taking several hours to complete and test. Most of the time was spent studying the codebase to identify the relevant entry points. Nevertheless, our success with fixing `PasswdSafe` demonstrates that patching a sizable program to improve its password retention behavior is straightforward.

**#5: System Processes.** Next, we tested lockscreen processes (`SystemUI` and `LocalSettingsService`) which leave PINs in memory (Table IV). 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$^{\dagger}$ in Table IV 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 did not 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. First, we modified Sample 1 to avoid the use of password strings, creating Sample 1$^{\dagger}$. With only this change, the derived secret is simply a hash of the password, retaining a variety of security vulnerabilities. We further 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. Note that only 30 lines are added in the client code for implementing both registration and login routines. The server requires additional modification on the database because it never sees user password or hash; instead, they have to store a crypt verifier and a salt value. (We're assuming the presence of a suitable SRP library on the server.)

Sample $1_{srp}$ shows the result of the fix: as before, all passwords are successfully erased 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 follows stronger cryptographic practices.

### C. 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 size of the necessary patch and the necessary effort to apply it 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 VI. Lastly, fixing an app "correctly" requires the adoption of a cryptographic 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.

## VIII. Discussion

**Stronger threat models.** Our solution successfully clears passwords right after login, providing an effective defense against sophisticated 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 they might be able to perform real-time monitoring of touchscreen activities to capture the password. This is outside the scope of this current paper. A possible defense, however, may be to leverage TrustZone for protection, similar as how the Android framework protects fingerprint data from attackers. In future work, we 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, moving password-management functionality out of the TextView class entirely.

**Credentials derived from passwords.** Our paper focuses on user passwords as they are of paramount importance, and as our paper shows, protecting passwords itself is an unresolved problem in Android. Credentials derived from passwords, such as hash values or cryptographic keys, are equally important but beyond the scope of our current paper. Obviously, these credentials and keys must also be deleted in a timely manner. We previously looked at SSL/TLS session key retention [34], discovering a number of issues similar to what we have found with password retention in this study.

**Fixes without modifying the Android framework.** Our current SecureTextView and KeyExporter are implemented as a set of patches to the AOSP `TextView` widget. Needless to say, this means that it's difficult or impossible to install on most users' phones. We considered the possibility of making these into standalone code which could be distributed as a separate library, allowing individual app authors to adopt our solution without waiting for Google. A standalone implementation would require duplicating a substantial amount of code from `TextView`, to ensure that we had proper behavior under all conditions (e.g., different layout styles, different input languages, and/or different Android versions).

We examined the possibility of simply subclassing the existing `TextView` widget, but our patches require changing how it stores and manages passwords; we cannot achieve the necessary memory erasure behaviors while also reusing the stock `TextView` widget class. While this is less than ideal, we would prefer for Google to make central changes to help app authors migrate to more secure password management behaviors.

**Fixes without modifying the apps.** Since several root causes of password retention are due to developers' insecure practices, our fixes require the insecure apps to be modified to address these problems. Concretely, we provide APIs that can help developers fix the apps without heavy code changes. It might be possible to develop fixes that do not require app modification, e.g., by tracking all password usage and cleaning them at the OS level. However, this would be very intrusive to the Android framework and cause considerable performance overhead. We believe that the fixes we have developed could present an easier path for adoption by app developers and the Android community.

**Usability.** As developers prioritize functionality over security [43], 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 [26], [42]. KeyExporter follows this principle that simplicity promotes security [1], and exports key materials in a similar fashion as how existing widgets export passwords.

**Centralized security management.** By shifting the responsibility of password management from app developers to KeyExporter and SecureTextView, 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 SecureTextView/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.

**Disclosure process.** We reported the issues in this paper to all of the impacted vendors in December 2018 via their standard security vulnerability reporting channel. For Google, we reported the problems in the AOSP keyboard, the lockscreen processes, and the Android widget implementation, and recommended that they update the official documentation about password protection best practices. We also contacted the keyboard app developers and the OAuth team at Facebook. Finally, we reached out to the app developers of the password managers with proposed fixes. The source code of our solution and patches are publicly available[5]. We provide KeyExporter as a standalone library, and public code samples to demonstrate how developers can harden their apps.

[5]https://github.com/friendlyJLee/totalrecall

**Static analysis.** Google regularly adds static analysis features to the Android Studio development environment to highlight undesirable coding practices. If Android were to adopt more secure alternatives to the password entry widget such as ours, the development environment could highlight uses of the default `TextView` widget and generate suitable warnings. Moreover, a campaign by Google to improve developers' code could certainly move its app ecosystem toward better practices for managing passwords. In the future, after our findings are widely disseminated, we can also build static analyzers to identify whether or not Android apps are adopting best practices in managing passwords.

## IX. Related Work

### A. Protecting sensitive data

Insecure data deletion has been an issue in desktops and servers for over a decade [13], and it is one of the fundamental causes of data exposure. Researchers have developed many solutions to address this. Chow et al. handle this by secure deallocation [14], Pridgen et al. aim at reducing encryption keys retained in the Java heap for desktops and servers [49], Dunn et al. use ephemeral channels where data will be securely erased after a session finishes [18], and Lee and Wallach study the retention of TLS secrets in Android memory [34]. Different from existing work, our paper 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. The example secure locations considered by existing work include cloud storage [57], CPU registers [40], and separate "trusted" CPU features like ARM TrustZone [65]. These proposals generally leverage features that are not universally available, and possibly also entail significant performance issues.

Researchers have also considered protecting sensitive data by detecting malicious application misbehaviors. Dynamic analysis techniques, such as data-flow analysis [20] and password-tracking [15], have been introduced to detect data leakage from applications. Static analysis techniques [4], [6], [12] have also been used to detect malicious behaviors. K-Hunt [35], for example, can pinpoint insecure cryptographic usage issues in software, including poor key sanitization.

Automated approaches facilitate scalable reasoning about security issues, but most are limited to analyses of the Java software stack, without necessarily looking at native methods in C, kernel buffers, or unreachable buffers that have not yet been garbage collected. In short, these tools are excellent when looking for a known vulnerability pattern, but are less useful in a case like ours, where we don't initially know anything other than the password string we're looking to find, wherever it may be.

### B. Memory forensics

In terms of memory acquisition techniques, Sylve et al. [56] first suggested a technique for capturing the physical memory of Android devices. In later work, Müller and Spreitzenbarth demonstrate that cold-boot attacks are feasible on Android phones [41]. Yang et al. [64] designed an acquisition technique

when the Android device is in firmware update mode. We used the system developed by Sylve et al. to perform our study.

Regarding memory analysis techniques, signature-based frameworks [47], [60] have been widely used by researchers to analyze memory dumps on different platforms. Various techniques have also been proposed to recover data structure from memory dumps using static analysis [10], dynamic analysis [17], and probabilistic analysis [37].

Memory forensics has been applied to the Android platform. A line of previous work has focused on extracting sensitive data from applications [3], [30], [58], [59]. Researchers have also looked at techniques to recover data beyond raw memory dumps, including the timeline of user activities [7] and GUI activities [52], [53].

### C. Security flaws in Android apps

Existing work has studied security flaws in Android apps, and revealed that developers have misused TLS library [21], [22], [25], cryptographic APIs [19], OAuth protocols [11], and fingerprint APIs [8]. Reaves et al. [51] analyzed mobile banking apps, reporting information leakage in these apps. Recent usability studies have also looked at why developers make mistakes, by analyzing the patterns of misuse [1], [2], [23], [42], [43].

Password managers have attracted particular attention because they directly handle sensitive passwords. Fahl et al. [22] revealed that many password managers are vulnerable to clipboard sniffing attacks. Silver et al. [55] found critical flaws in auto-fill functionality. Li et al. [36] found problems in web-based password managers.

## X. Conclusion

In this paper, we have performed a comprehensive study on password retention in Android. Our analysis techniques—searching through memory dumps—proved to be robust and effective at discovering problems and validating our alternatives. We found problems with the core Android platform, as well as a wide variety of popular apps, including keyboard input apps. We developed suitable patches for Android's TextView widget to address these problems, assisting apps to follow more secure password management practices with only modest changes in their code.

**Acknowledgement:** We thank our shepherd, Adam Aviv, and the anonymous reviewers for their valuable feedback. This work was supported in part by NSF grants CNS-1801884, CNS-1409401, and CNS-1314492.

## References

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic APIs," in *38th IEEE Symposium on Security and Privacy (S&P '17)*. IEEE, 2017.

[2] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *37th IEEE Symposium on Security and Privacy (S&P '16)*, 2016.

[3] D. Apostolopoulos, G. Marinakis, C. Ntantogian, and C. Xenakis, "Discovering authentication credentials in volatile memory of Android mobile devices," in *Conference on e-Business, e-Services and e-Society*. Springer, 2013.

[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket." in *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[5] N. Artenstein, "BROADPWN: Remotely compromising Android and iOS via a bug in Broadcom's WiFi chipsets," in *Black Hat USA*, 2017.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '14)*, 2014.

[7] R. Bhatia, B. Saltaformaggio, S. J. Yang, A. Ali-Gombe, X. Zhang, D. Xu, and G. G. Richard III, "Tipped off by your memory allocator: Device-wide user activity sequencing from Android memory images," in *Network and Distributed System Security Symposium (NDSS '18)*, 2018.

[8] A. Bianchi, Y. Fratantonio, A. Machiry, C. Kruegel, G. Vigna, S. P. H. Chung, and W. Lee, "Broken fingers: On the usage of the fingerprint API in Android," in *Network and Distributed System Security Symposium (NDSS '18)*, 2018.

[9] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *33rd IEEE Symposium on Security and Privacy (S&P '12)*, 2012.

[10] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *16th ACM Conference on Computer and Communications Security (CCS '09)*, 2009.

[11] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *21st ACM Conference on Computer and Communications Security (CCS '14)*, 2014.

[12] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *9th international conference on Mobile systems, applications, and services (MobiSys '11)*. ACM, 2011.

[13] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *13th USENIX Security Symposium (Security '04)*, 2004.

[14] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," in *14th USENIX Security Symposium (Security '05)*, 2005.

[15] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati, "SpanDex: Secure password tracking for Android." in *23rd USENIX Security Symposium (Security '14)*, 2014.

[16] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse." in *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[17] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *16th ACM Conference on Computer and Communications Security (CCS '09)*, 2009.

[18] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel, "Eternal Sunshine of the spotless machine: Protecting privacy with ephemeral channels." in *USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, 2012.

[19] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *20th ACM Conference on Computer and Communications Security (CCS '13)*, 2013.

[20] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[21] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in *19th ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[22] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith, "Hey, you, get off of my clipboard," in *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 144–161.

[23] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy&paste on android application security," in *38th IEEE Symposium on Security and Privacy (S&P '17)*. IEEE, 2017.

[24] D. Florencio and C. Herley, "A large-scale study of web password habits," in *16th international conference on World Wide Web (WWW '07)*. ACM, 2007.

[25] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *19th ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[26] M. Green and M. Smith, "Developers are not the enemy! The need for usable security APIs," *IEEE Security & Privacy*, vol. 14, no. 5, pp. 40–46, 2016.

[27] S. Gunasekera, *Android Apps Security*. Apress, 2012.

[28] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," in *17th USENIX Security Symposium (Security '08)*, 2008.

[29] R. Hay, "Undocumented patched vulnerability in Nexus 5X allowed for memory dumping via USB," *Security Intelligence*, 2016. [Online]. Available: https://ibm.co/Bdeidu

[30] C. Hilgers, H. Macht, T. Muller, and M. Spreitzenbarth, "Post-mortem memory analysis of cold-booted Android devices," in *Eighth International Conference on IT Security Incident Management & IT Forensics*. IEEE, 2014.

[31] B. Kaliski, "PKCS #5: Password-based cryptography specification version 2.0," Tech. Rep., 2000, https://tools.ietf.org/html/rfc2898.

[32] J. Kelsey, B. Schneier, C. Hall, and D. Wagner, "Secure applications of low-entropy keys," in *International Workshop on Information Security*. Springer, 1997, pp. 121–134.

[33] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P '19)*, 2019.

[34] J. Lee and D. S. Wallach, "Removing secrets from Android's TLS," in *Network and Distributed System Security Symposium (NDSS '18)*, 2018.

[35] J. Li, Z. Lin, J. Caballero, Y. Zhang, and D. Gu, "K-Hunt: Pinpointing insecure cryptographic keys from execution traces," in *25th ACM Conference on Computer and Communications Security (CCS '18)*, 2018.

[36] Z. Li, W. He, D. Akhawe, and D. Song, "The emperor's new password manager: Security analysis of web-based password managers." in *23rd USENIX Security Symposium (Security '14)*, 2014.

[37] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures," in *Network and Distributed System Security Symposium (NDSS '11)*, 2011.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (Security '18)*, 2018.

[39] N. Mavrogiannopoulos and J. Simon, "GnuTLS," 2003, https://gnutls.org/.

[40] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *20th USENIX Security Symposium (Security '11)*, 2011.

[41] T. Müller and M. Spreitzenbarth, "FROST: Forensic recovery of scrambled telephones," in *International Conference on Applied Cryptography and Network Security (ACNS '13)*, 2013.

[42] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do Java developers struggle with cryptography APIs?" in *38th International Conference on Software Engineering (ICSE '16)*. ACM, 2016.

[43] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith, "Why do developers get password storage wrong?: A qualitative usability study," in *24th ACM Conference on Computer and Communications Security (CCS '17)*, 2017.

[44] OpenSSL Software Foundation, "OpenSSL," 1999, https://www.openssl.org/.

[45] Oracle, "Java cryptography architecture (JCA) reference guide," 2003, https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html.

[46] C. Percival and S. Josefsson, "The scrypt password-based key derivation function," Internet Requests for Comments, RFC Editor, RFC 7914, Aug. 2016, https://tools.ietf.org/html/rfc7914.

[47] N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh, "FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digital Investigation*, vol. 3, no. 4, pp. 197–210, 2006.

[48] T. Pornin, "BearSSL," 2017, https://bearssl.org/.

[49] A. Pridgen, S. L. Garfinkel, and D. S. Wallach, "Present but unreachable: Reducing persistent latent secrets in HotSpot JVM," in *50th Hawaii International Conference on System Sciences (HICSS '17)*, 2017.

[50] N. Provos and D. Mazieres, "A future-adaptable password scheme." in *1999 USENIX Annual Technical Conference, FREENIX Track*, 1999.

[51] B. Reaves, N. Scaife, A. Bates, P. Traynor, and K. R. Butler, "Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world," in *24th USENIX Security Symposium (Security '15)*, 2015.

[52] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "GUITAR: Piecing together Android app GUIs from memory images," in *22nd ACM Conference on Computer and Communications Security (CCS '15)*, 2015.

[53] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. Richard III, "Screen after previous screens: Spatial-temporal recreation of Android app displays from memory images," in *25th USENIX Security Symposium (Security '16)*, 2016.

[54] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor, "Encountering stronger password requirements: user attitudes and behaviors," in *6th Symposium on Usable Privacy and Security (SOUPS '10)*. ACM, 2010, p. 2.

[55] D. Silver, S. Jana, D. Boneh, E. Y. Chen, and C. Jackson, "Password managers: Attacks and defenses." in *23rd USENIX Security Symposium (Security '14)*, 2014.

[56] J. Sylve, A. Case, L. Marziale, and G. G. Richard, "Acquisition and analysis of volatile memory from Android devices," *Digital Investigation*, vol. 8, no. 3, pp. 175–184, 2012.

[57] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "CleanOS: Limiting mobile data exposure with idle eviction," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, 2012.

[58] V. L. Thing, K.-Y. Ng, and E.-C. Chang, "Live memory forensics of mobile phones," *digital investigation*, vol. 7, pp. S74–S82, 2010.

[59] R. J. Walls, E. G. Learned-Miller, and B. N. Levine, "Forensic triage for mobile phones with DEC0DE," in *20th USENIX Security Symposium (Security '11)*, 2011.

[60] A. Walters, *The Volatility framework: Volatile memory artifact extraction utility framework*, 2007, https://github.com/volatilityfoundation/volatility.

[61] T. D. Wu, "The secure remote password protocol." in *Network and Distributed System Security Symposium (NDSS '98)*, 1998.

[62] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan, "NDroid: Toward tracking information flows across multiple Android contexts," *IEEE Transactions on Information Forensics and Security*, 2019.

[63] L.-K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis." in *23rd USENIX Security Symposium (Security '14)*, 2012.

[64] S. J. Yang, J. H. Choi, K. B. Kim, R. Bhatia, B. Saltaformaggio, and D. Xu, "Live acquisition of main memory data from Android smartphones and smartwatches," *Digital Investigation*, vol. 23, pp. 50–62, 2017.

[65] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-assisted secure execution on ARM processors," in *37th IEEE Symposium on Security and Privacy (S&P '16)*, 2016.