Removing Secrets from Android's TLS

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

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

I. Introduction

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

However, unless these secrets are deleted properly after a session completes, they reside in memory, and thus become vulnerable to memory disclosure attacks, allowing recorded communications to be subsequently decrypted. Attack vectors vary including physical techniques such as "cold boot attack" [12], which physically extract memory chips, and throughout software exploitations like the OpenSSL Heartbleed vulnerability (CVE-2014-0160) which exposes sensitive data in memory to remote attackers without any privileges.

A variety of tactics are used on TLS to ensure secrets are quickly forgotten. For example, modern TLS cipher suites support perfect forward secrecy (PFS), ensuring that key material which must be saved over a long period cannot be used to decrypt previous sessions that an attacker may have recorded. According to SSL Labs's monitoring[1], about 89% of HTTPS websites support PFS in August 2017, versus only 46% in October 2013.

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

Several recent studies have featured the recover of cryptographic key material as part of a forensic examination. Taubmann et al. [31] provide a technique to extract master secrets at runtime using virtual machine introspection techniques, and Kambic [15] provides a similar analysis of Windows systems. Pridgen et al. [22] investigate Java TLS implementations, finding key material remaining in memory as a consequence of the JVM's garbage collector. A copying garbage collector may leave multiple dead copies of a key behind in memory that are not "reachable" as live data, yet are still vulnerable to forensic extraction.

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

Several recent Android vulnerabilities underscore the practicality of memory disclosure attacks. For example, a recent vulnerability in a Broadcom WiFi chipset [3] allowed an attacker to take control of the WiFi chip, using it to conduct arbitrary reads and writes into the main CPU's memory. And of course, once an attacker has physical access to a device, they may have access to further vulnerabilities that allow memory to be dumped (see, e.g., this Nexus 5X issue [13]).

Of course, users of any smartphone device may connect to the Internet over unencrypted WiFi hotspots, allowing attackers to record their communications. Even WiFi's WPA2 encryption scheme has vulnerabilities [33], enabling adversaries to eavesdrop on supposedly safe WiFi systems. If a phone or computer is using TLS for all its connections, then these WiFi issues are less of a concern, since an eavesdropper would still see only encrypted traffic.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23365
www.ndss-symposium.org

[1] See https://www.ssllabs.com/ssl-pulse/

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

We started our research with two hypotheses:

- The combination of multiple software layers that implement the TLS protocol, along with Java's garbage collection system, provide opportunities for subtle bugs in key management.
- Many Android applications will fail to manage cryptographic key material lifetime alongside the Android application lifecycle.

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

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

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

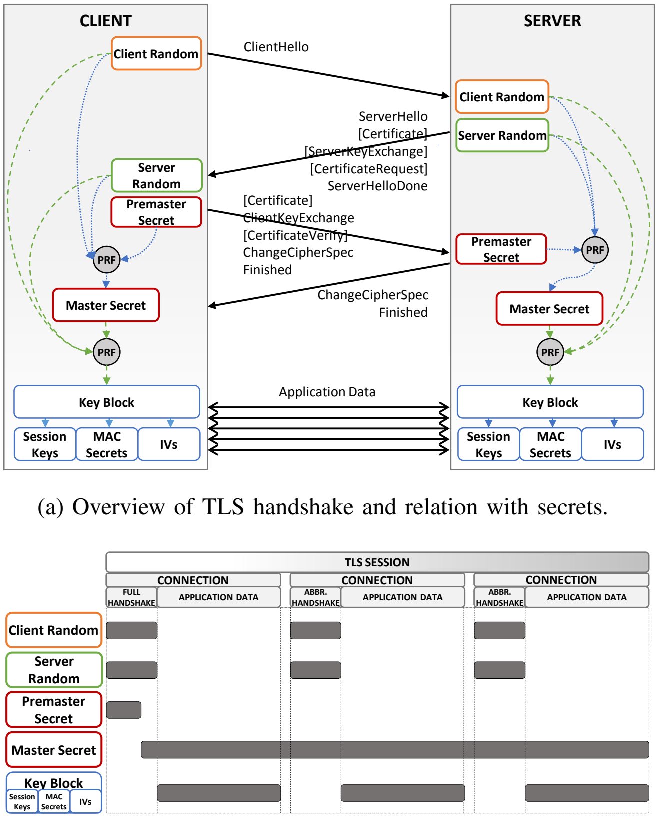
The rest of the paper is organized as follows. Section II gives background on TLS concepts and architecture on Android. Section III provides our black-box analysis method and design details of our automated framework. In Section IV, in-depth analysis results are described in detail. Section V evaluates how this problem is exploitable for attacks in practice. The solutions to address this issue are discussed in Section VI. We introduce related work in Section VII and conclude the paper in Section VIII.

II. Background

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

A. TLS

1) TLS Handshake and Secrets: Figure 1a abstracts the TLS handshake protocol, described in full detail in its RFC [8]. For now, we focus only on how secrets are created and shared. During a handshake, five artifacts are generated or calculated and shared between a client and a server. 32 bytes of client and server randomness are generated by a client and a server, respectively, and shared as a plain text through "hello" messages. Those values are used to calculate a master secret and a key block. A pre-master secret is then shared throughout the next KeyExchange messages, using RSA or Diffie-Hellman key exchange. In RSA key exchange mode, it is the client's role to generates a 48-byte random value for the pre-master secret. Then, the client encrypts it using the server's public key and sends to the server. In the Diffie-Hellman scheme, the pre-master secret is derived by mixing generated DH public-private pairs with the peer's DH public value. Whether using RSA or Diffie-Hellman, both sides end up sharing the pre-master secret, while an eavesdropper cannot derive this value.



(a) Overview of TLS handshake and relation with secrets.

(b) Minimum effective lifetime of secrets.

Fig. 1: TLS handshake and secrets.

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

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

2) Session Resumption: TLS provides the notion of "session resumption" which allows an abbreviated handshake, avoiding the computationally expensive RSA or Diffie-Hellman handshakes, and reusing the previously shared master

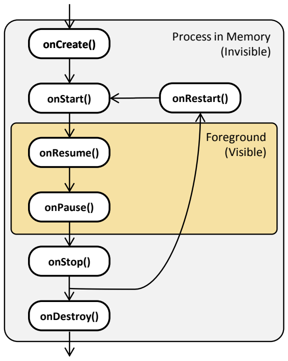


Fig. 2: Android application lifecycle.

secret. This necessarily implies that the master secret must survive beyond the lifetime of any one network connection in order to be reused by a subsequent connection.

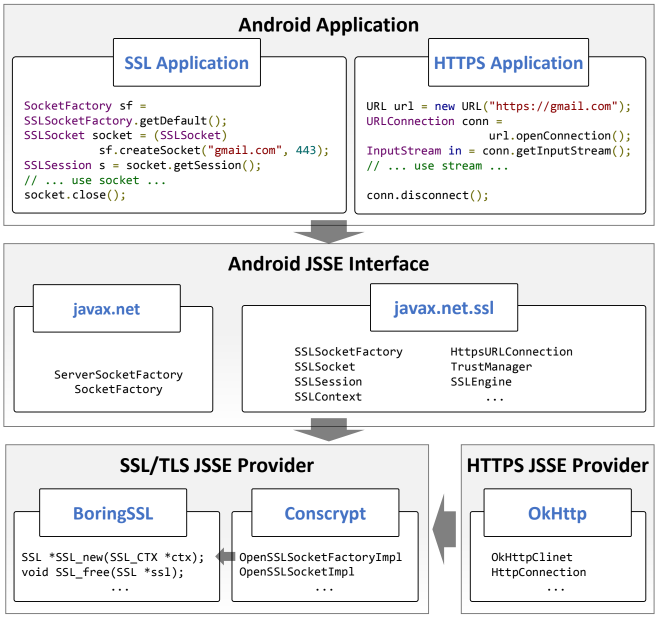
Session resumption was supported from the very beginning of SSL through session identifiers [8] and later through session tickets [25]. With session IDs, both the client and server maintain copies of their master key. The client presents the appropriate session ID, and the server can accept or reject that ID. With the newer session tickets, the server no longer needs to preserve its copy of the master key. Instead, the session ticket, stored on the client, has all the necessary state. The ticket is appropriately encrypted to make it safe to send in the clear. The benefit of this scheme, particularly in large server clusters, is to simplify the server's need to preserve and replicate state.

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

3) Lifetime of TLS Secrets: Figure 1b summarizes the minimum possible lifetime of secrets generated during the handshake when a TLS session is shared across multiple connections. The master secret has the longest minimum lifetime since everything else is unnecessary once any given connection is closed, while the master secret is reused across connections. RFC 5246 specifically recommends that a master secret be maintained for no longer than 24 hours. In our own investigations, we have observed that modern web servers like Apache and Nginx will expire a master secret after only 5 minutes. This implies that a client can and should delete its master secrets in a similar timeframe.

B. Android application lifecycle

One notable distinction between Android applications and traditional desktop programs is the *Android application lifecycle* shown in Figure 2. With a traditional desktop operating system, a user will launch a program, then it is loaded into memory, and it will remain running until the user explicitly closes the program. Android operates differently, reserving the right to kill off an application at any time when system resources are exhausted. Applications can also be "paused" and "resumed" by a user, or even "stopped" and "restarted". These distinctions matter. A "paused" application might still be visible to the user in a multi-window scenario, while a "stopped" application is no longer seen and it's a good practice at this point to close active connections and return resources back to the system.



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

```
URL url = new URL("https://gmail.com");
URLConnection conn =
                  url.openConnection();
InputStream in = conn.getInputStream();
// ... use stream ...

conn.disconnect();
```

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

Fig. 3: Android JSSE architecture.

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

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

C. TLS Implementation on Android

Android provides HTTPS and TLS implementations through the Java Secure Socket Extension (JSSE) API. The JSSE model is based on a "provider" architecture, providing implementation independence and algorithm extensibility. Figure 3 sketches the full stack of JSSE components on recent versions of Android.

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

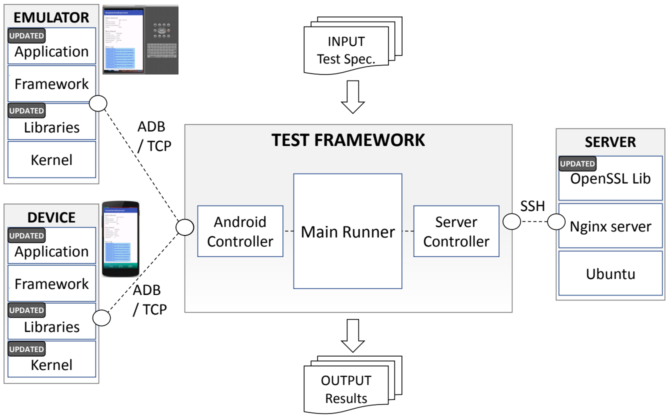


Fig. 4: The overview of the analysis framework.

III. Black-box security analysis

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

A. Methodology

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

Our black-box analysis framework, outlined in Figure 4, consists of Android devices with memory acquisition features, an HTTPS web server, and a test framework. Our custom Android application makes TLS connections using standard JSSE APIs. We use an instrumented web server which can record all of the key material that it sees. Using a variety of test scripts, we can run our client and server in many different configurations. Once complete, we dump the client's memory and the server's key material. We built tools to automatically search through the client memory dumps for this key material.

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

B. Test Framework

There have been several studies on Android, comparably dumping memory to look for sensitive data [2], [17], [1], [20], [37]. However, all of these approaches required a manual approach to capture and search the relevant memory images. The problem with manual approaches is that they don't easily scale to examining hundreds of application runs. For example, Apostolopoulos et al. [2] spent six months examining thirty Android applications. Furthermore, if the effects being measures are probabilistic in nature, multiple runs will be necessary, further burdening the data collection and analysis process. Ntantogian et al. [20], for example, studied application lifecycle issues with credential usage, taking three months to examine 390 test-cases for thirteen applications.

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

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

C. Supporting memory dump on Android devices

Our threat model assumes memory disclosure attacks, and we need to set up that situation on a real Android device as well as an emulator. Android devices do not provide memory dumping as a native feature, requiring us to rebuild the Android kernel to include the LiME kernel module [29], giving the necessary functionality. For contrast, when running Android under the QEMU virtual machine emulator, we can use the `pmemsave` command which does exactly what we need without requiring a custom kernel.

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

D. HTTPS client on Android devices

We built simple HTTPS / TLS clients for Android to exercise the standard cryptographic libraries. Recent research [23] shows 84% of applications use standard libraries for TLS; our approach allows us to most efficiently exercise the standard Android's HTTPS / TLS software stack.

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

E. Server

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

The only necessary customizations on our server are to log all of the cryptographic keys used in the OpenSSL library. As with our customized Android environment, this web server

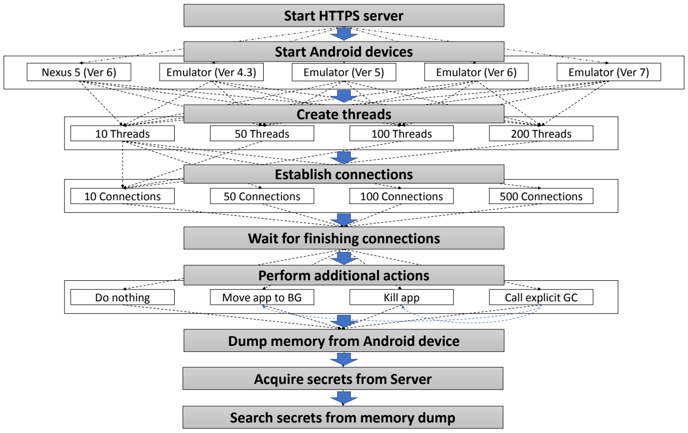


Fig. 5: The experiment scenario.

would be unsuitable for use in a production environment, since a log of every cryptographic key used would, in a production environment, represent an unacceptable security vulnerability. After all, the whole point is to forget unnecessary key material! Nonetheless, we want to capture these keys so we know what to search for in our client memory dumps.

F. Experiment

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

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

Overall, this round of experiments spanned three weeks of effort to capture more than 200 different test cases, with a fair bit of our time spent ironing out bugs in our framework versus bugs in Android itself. We contrast this with Ntantogian et al. [20], who spent three months to evaluate 390 cases for 13 applications.

G. Results

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

Threads	Conn.	Total Conn.	Nexus	Emul 4.3	Emul 5	Emul 6	Emul 7
10	10	100	8	5	5	5	6
10	50	500	29	6	5	7	5
10	100	1000	16	5	5	6	6
10	500	5000	11	5	5	6	6
50	10	500	6	5	5	7	6
100	10	1000	12	6	6	6	5
200	10	2000	85	5	5	6	5

(a) Baseline measurements, varying the degree of concurrency[1].

Devices	Nothing	GC	BG	Kill	GC → BG	GC → Kill
Nexus	29	5	5	3	5	3
Emul 4.3	6	5	5	6	5	5
Emul 5	5	4	5	0	5	2
Emul 6	7	5	5	2	5	1
Emul 7	5	5	5	5	5	2

(b) After 500 connections and other additional actions[1].

Application	Nothing	BG	Kill	1hour → BG	BG → 1hour
HTTPS App[1]	5	5	5	0	5
TLS App[1]	5	0	0	0	0

(c) HTTPS APIs vs. TLS APIs

Application	Nothing	BG	Kill	1hour → BG	BG → 1hour
HTTPS App[2]	1	1	1	1	1
TLS App[2]	1	1	1	1	1
Chrome	1	1	1	1	1

(d) Android Chrome vs. our test application

TABLE I: Surviving master keys under various configurations.

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

We also note that none of the remaining master secrets that we found were duplicates. Each one represents a distinct key. This suggests that we don't have a problem originating from Java's copying garbage collector or careless object copying.

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

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

Additionally, we note that even killing off our application does not eliminate its keys from memory. Since the OS process is dead, this suggests that Android is lazy about zeroing out such memory. Of course, memory is a scarce resource in a

[1] Creating a new SSLContext for each connection, so a full handshake on every connection.

[2] Using the default SSLContext to support the session resumption.

mobile phone and it will *eventually* be reused and cleared, but our results suggest the kernel could zero out dead process memory more aggressively.

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

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

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

H. Observation and Raised Questions

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

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

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

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

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

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

Question 5. *Why do secrets last longer when using the default SSLContext rather than when creating new contexts for each connection?*

IV. In-depth Analysis of Android Framework

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

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

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

A. Overview

We summarize the relation among the three modules in Figure 6. There are three key concepts of TLS implementations that are important to understand: ***Context***, ***Connection***, and ***Session***. A context encapsulates the TLS implementation itself, including the implementation's configuration and supported cipher suites. A context is typically shared across every TLS connection. Importantly, contexts are responsible for caching session state for fast session resumption. A connection represents a single TLS connection, typically corresponding to a specific TCP/IP socket session. Every connection has a corresponding session and context within which it's defined and which manages its long-term state. Once the TCP/IP socket closes, the connection is done. Conversely, a session represents the TLS state that may be resumed in an abbreviated handshake. Sessions are also associated with contexts. When a TCP/IP socket closes, the session state, as stored in the context, will survive.

Those three concepts are supported by the `SSLContext`, `SSLSocket`, and `SSLSession` classes in the `javax.net.ssl` package in JSSE. Additionally, JSSE provides an `SSLSocketFactory` class that encapsulates the `SSLContext` class, providing a convenient wrapper for creating many sockets without having to juggle the `SSLContext` objects. Since the `javax.net.ssl` package only contains interfaces, corresponding concrete classes exist both in Conscrypt and BoringSSL as shown in Figure 6. For the `SSLContext` class, the `OpenSSLContextImpl` and `SSLParametersImpl` classes in Conscrypt provide the glue routines between Java and C, and the `SSL_CTX` structure in BoringSSL contains the *actual* TLS context.

Conscrypt's `SSLParametersImpl` class is important because it contains the `ClientSessionContext` and `ServerSessionContext` classes which are responsible for tracking previous sessions. In turn, Conscrypt's `OpenSSLSocketImpl` class corresponds to the `SSL` structure in BoringSSL, supporting JSSE's interfaces for `SSLSocket`. And likewise, Conscrypt's `OpenSSLSessionImpl` corresponds to BoringSSL's `SSL_SESSION` structure, which underpins JSSE's `SSLSession` class.

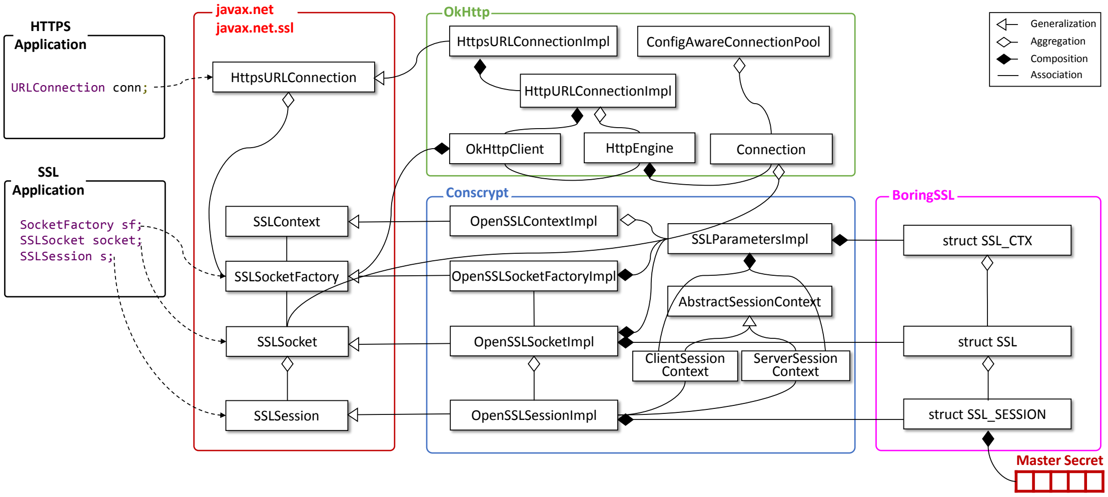


Fig. 6: Relationship between HTTPS and TLS implementations on Android.

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

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

B. BoringSSL

BoringSSL provides the full functionality of TLS by itself, and can be used by C programs without interacting with any of the Java modules. Subroutines for handshaking and managing secrets are all implemented in BoringSSL. The pre-master secret is deleted properly after finishing handshake. Every master secret is allocated and located within an SSL_SESSION object after establishing a session, and it is cleaned when no other object retains a reference to this SSL_SESSION object. Because BoringSSL is implemented in C, the programming language runtime system provides no assistance in detecting when an SSL_SESSION is dead. Instead, BoringSSL supports manual reference counting for its data structures. All major data structures have a reference count field, and clients of BoringSSL are responsible for issuing calls to increment or decrement these counts. In typical usage, however, the counter is initialized to one when the data structure is created, and each free-function (e.g., SSL_SESSION_free()), will decrement the reference counter. If the free-function finds the reference counter is zero, it will then “cleanse” (i.e., zeroize) the memory region before handing the memory back to the memory allocator. If, however, the reference count is non-zero, then the free-function will only decrement the counter and otherwise do nothing else. This logic is shown in Figure 7.

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

Fig. 7: SSL_SESSION_free function in BoringSSL.

When BoringSSL is used on its own, these reference counts appear to be managed correctly. For example, when an SSL structure is created for a TLS connection, it can be initialized with an existing SSL_SESSION or with null. In the latter case, it creates a new SSL_SESSION, with a reference count of 1 and performs the full TLS handshake. In the former case, it increments the reference count on the SSL_SESSION with which it has now been associated.

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

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

C. *Conscrypt*

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

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

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

Conscrypt's session cache supports TLS session resumption, but we found it creates problems in releasing keys. Figure 8 includes three problematic codes that contribute to keeping master secrets unnecessary long in the memory without any performance benefit. (This answers Question 1 in III-H. Secret retention seems to be the result of a bug, not a performance trade-off.)

1) Depending on GC: One of the main causes of master secret retention is that Conscrypt places critical code in its finalize() method as shown in Figure 8a. In Java, the finalize() method is only invoked when

```
public class OpenSSLSessionImpl implements
    SSLSession {
 ...
 @Override
 protected void finalize() throws Throwable {
   try {
     if (sslSessionNativePointer != 0) {
       NativeCrypto.SSL_SESSION_free(sslSessionNativePointer);
     }
   } finally {
     super.finalize();
   }
 }
}
```

(a) finalize() method on OpenSSLSessionImpl.

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

(b) Removal routine in AbstractSessionContext.

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

 ...
}
```

(c) The default constructor in OpenSSLContextImpl.

Fig. 8: Excerpts of session management code from Conscrypt on Android 7.1.

garbage collection is triggered due to insufficient heap memory in JVM. This means it can take quite a while, after an `OpenSSLSessionImpl` becomes garbage before its `finalize()` method might be called. This, in turn, means that Conscrypt will keep the underlying BoringSSL session state, including the master secret, live well beyond when it should have been cleansed.

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

2) LRU implementation of the session cache: Conscrypt maintains the session caches holding `SSLSession` objects for session resumption in its `ClientSessionContext` and `ServerSessionContext` classes. The ideal deletion logic would be to remove session objects after a designated time, such as ten minutes or perhaps one hour. Unfortunately, those classes do not provide such an explicit deletion routine. Instead, as the code in Figure 8b shows, the session cache is designed to work in an LRU fashion (see the code dealing with `removeEldestEntry()`). Removing the eldest `SSLSession` only happens when a new `SSLSession` is added to the session cache, and the session cache is full. Thus, even though the master secrets may have expired, they will only be removed as a result of the LRU cache's eviction logic. This lazy deletion allows attackers to get as many master secrets as there exist slots in the session cache; the default size of the session cache is 10 and 100 for a client and a server, respectively.

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

3) Singleton SSLContext: Most of the root classes in Figure 6 are created using a "singleton" pattern and managed globally, so there is only ever one instance of these Java classes. This prevents `SSLSession` objects from ever being garbage collected. This issue also applies to the `ConfigAwareConnectionPool`, `SSLContext`, `SSLSocketFactory`, and `SSLParametersImpl` classes. The code in Figure 8c shows that the default `SSLContext` member is defined as private and static, and thus it is initialized once when `SSLContext` is first created with the default mode, and it will never become garbage. Consequently, while an application is running, all master secrets relating to the default `SSLContext` will never be removed. (This answers Question 5 in III-H. The default SSLContext never becomes garbage.)

D. OkHttp

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

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

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

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

E. Summary of the problem

BoringSSL itself correctly supports reference counts to track access to its internal structures, and will properly zeroize key material once the reference count goes to zero. OkHttp has a timer to detect expired entries in its connection pool. The biggest issue, however, is how all these layers interact with Conscrypt's thin Java wrapper around the BoringSSL library. In Conscrypt, cleanup depends on garbage collection to run, allowing master secrets to survive in memory long after they should be removed. Consequently, the number of master secrets in memory is constant—the size of the session cache.

(This answers Question 2 in III-H. The connection pool's size does not vary with the number of concurrent connections.)

- Using HttpsURLConnection with the default SSLContext: At least ten master secrets can be found during the whole application lifecycle once the application accesses more than ten sites.
- Using HttpsURLConnection with a disposable SSLContext (i.e., with session resumption *not* supported): At least five secrets can be found once the application accesses more than five sites until five minutes elapse and then GC is called. If the application is paused before those five minutes expire, the secrets will persist since there is no chance to call GC.
- Using SSLSocket with the default SSLContext: At least ten master secrets can be found during the whole application lifecycle once the application accesses more than ten sites.
- Using SSLSocket with a disposable SSLContext (i.e., with session resumption *not* supported): Some secrets remain after connections are completed, but they are deleted quickly when the application is paused or sent to the background.

V. Evaluation of Attack Feasibility

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

A. *Threat model*

We have two assumptions in our threat model. First, the attacker is able to passively capture network packets. This may occur over WiFi or any other network, and is quite feasible in practice. One recent study showed that packet capture from WPA2-encrypted WiFi is feasible [33], and of course, the attacker may control the network infrastructure.

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

Also, we do assume that attackers possess Android memory disclosure vulnerabilities, which can be physical memory dumping exploits such as the vulnerability in the Nexus 5X [13]. They also can be software-based exploits that allow them to access contents in memory remotely, such as the recent vulnerabilities in WiFi chipsets [3] and Bluetooth chipsets[2].

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

- The 48 bytes corresponding to a master secret must be extracted from the device's memory in a reasonable time.
- Master secrets must remain in the phone, while the phone is being actively used, rather than simply quiescent. Otherwise, memory needs from active apps will drive the Android system to reclaim memory from a quiescent app.

```
struct ssl_session_st {
  CRYPTO_refcount_t references;
  int ssl_version;
  ...
  uint32_t key_exchange_inf;

  int master_key_length;
  unit8_t master_key[SSL_MAX_MASTER_KEY_LENGTH];

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

Fig. 9: SSL_SESSION structure on Android 7.1.2.

B. *Extracting a master secret*

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

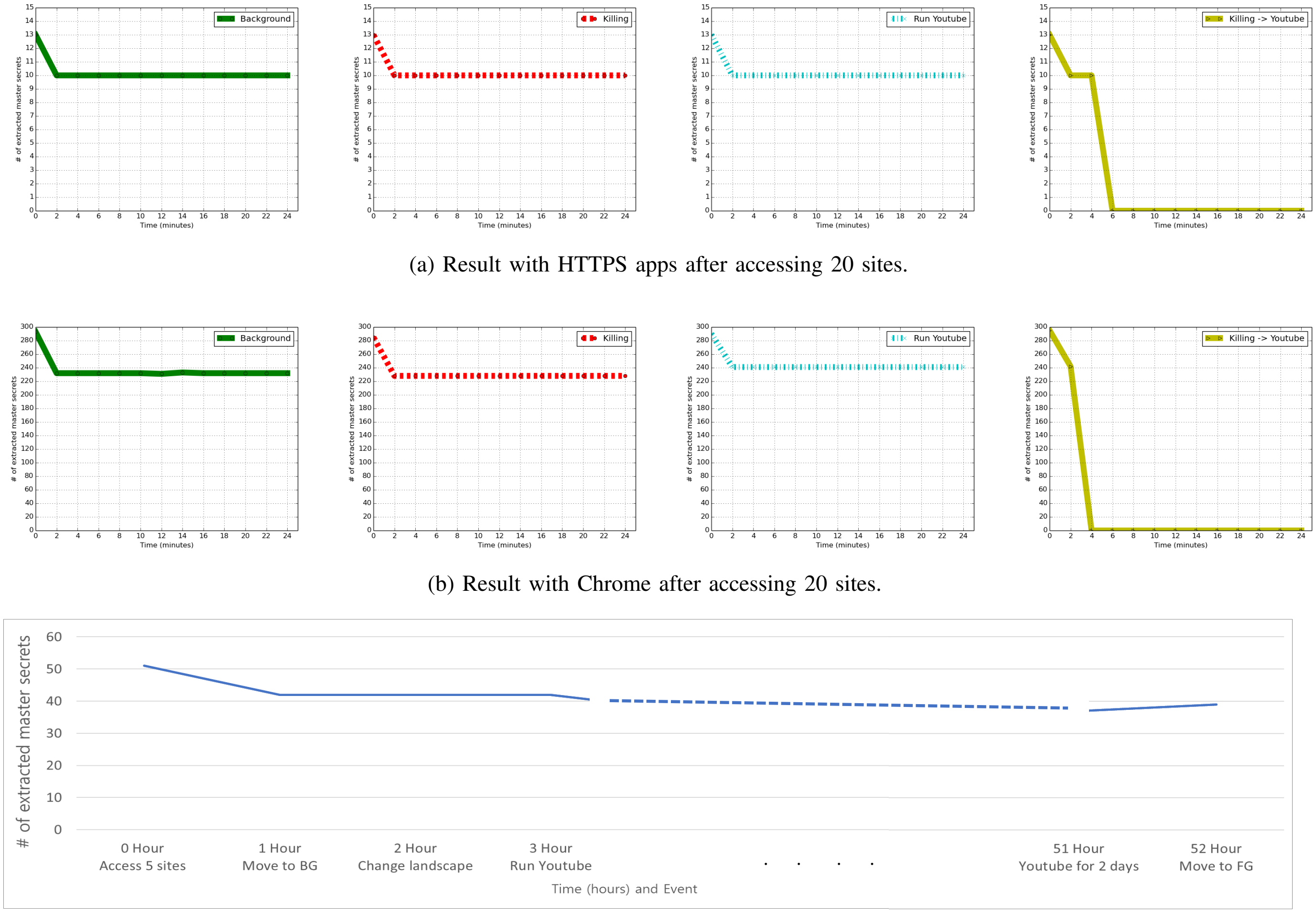
Luckily for attackers, the surrounding C structure creates patterns that are easy to recognize. Figure 9 shows the SSL_SESSION structure on Android 7. The master_key variable is located after ssl_version and master key length. Also, the secret is followed by a session_id_length variable. All those variables have well-defined values and they occupy 12 bytes in their specific positions. This signature pattern is sufficient for a rapid search through a memory image. We confirmed that we could extract all master secrets from gigabytes of a memory image in seconds, both from our HTTPS applications and from Android Chrome. (This also suggests that Android Chrome is based on a similar BoringSSL codebase.)

C. *Measuring secret retention in active use*

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

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

[2] See https://www.armis.com/blueborne/



(a) Result with HTTPS apps after accessing 20 sites.

(b) Result with Chrome after accessing 20 sites.

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

Fig. 10: Measuring secret retention after accessing popular sites.

The results are shown in Figures 10a and 10b for our application and Chrome, respectively. The Chrome results are particularly surprising. Note the size of the y-axis. We extracted *hundreds* of master secrets from Chrome, many minutes past their last use. About ten master secrets are accumulated in memory for each new site, without any deletion. This suggests that Chrome has made a performance-vs-security tradeoff, keeping key material live rather than allowing it to be deleted aggressively.

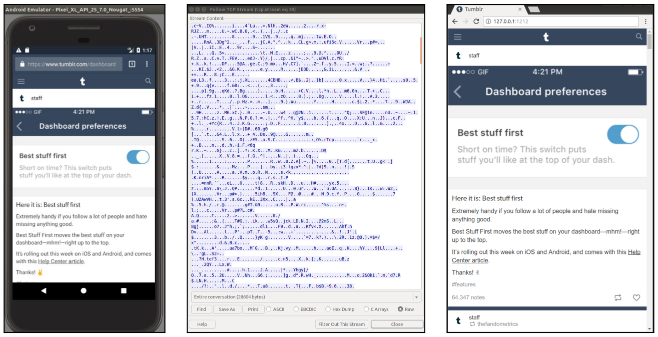
The only case where Android's memory management saved us was in the case when we killed the app or Chrome browser and subsequently ran YouTube. YouTube's memory demands forced Android to reclaim memory for YouTube, zeroing out the expired pages from our app.

Lastly, we conducted a long-term experiment for Chrome giving different events to see whether Chrome eventually cleans up its keys. We accessed five sites using Chrome and then left it in memory. We performed various activities every hour, including rotating the phone, running YouTube briefly, and even letting YouTube run for two days solid. (We chose the YouTube application because it alone creates non-trivial workload on the phone.) The result in Figure 10c clearly shows that no such events impact master secret retention in the long inactive application. YouTube alone failed to push the system to kill background applications on today's powerful devices though it constantly downloads data, decodes video, and displays it on screen.

Our results show that attackers have an unnecessarily large window of time to get master secrets from victims' phones. While "power user" phones might forcibly reclaim memory, many users don't use their phones so aggressively, and as a result, their TLS master keys will remain in memory.

D. Decrypting TLS communication

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



(a) Android 7. (b) Captured stream. (c) Recovered page.

Fig. 11: Rendering TLS communication successfully.

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

Figures 11a shows the snapshot of accessing a website using Chrome on the Android 7 emulator. All HTTPS communications attackers see are encrypted as Figures 11b. After accessing some sites, we used the phone as a normal user, making calls, and sending text messages. After leaving the phone locked for a day, we dumped the memory, modeling an attacker who physically captured the phone. As Figure 11c shows, our tool recovered the victim's web traffic. Furthermore, since decrypted HTTPS requests will often include usernames and passwords, or HTTP session cookies, such credentials will allow the attacker to impersonate the user to the site.

VI. Discussion

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

A. Solutions

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

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

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

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

2) Hooking Android's core framework: A better solution is to modify the Android core library to remove master secrets as part of the Android application lifecycle, ensuring that master secrets are cleaned up, regardless of the lifecycle state of an application. Done this way, our key retention problem can be addressed by triggering a zeroization routine when the application's lifecycle is changed.

As a prototype solution, we modified the Android Core Framework, hooking the Activity class to trigger the cleansing routine when the application is going to the background or being killed. Figure 12 shows our simplified routine applied to the Android framework. It has three steps. First, it removes all sessions in the session cache from singleton objects such as `SSLContext` and `SSLParameters`. While the JSSE interface does not explicitly expose an interface to clear the session cache, we found a hack to clear the session cache to use standard JSSE APIs. We clear the cache by resetting the cache size to zero and reverting it, triggering Conscrypt's lazy deletion routines. Though we delete the session in the session pools in Conscrypt, OkHttp also has connections in the connection pool that holds an `SSLSocket` object which in turn has `SSLSession`.

OkHttp provides `ConfigAwareConnection` object which is managed on Android as a singleton, so it can

```
$Contexts \leftarrow \emptyset$
$Objs \leftarrow \text{GetAllSingleton}(OpenSSLSocketFactoryImpl)$
$Objs \leftarrow Objs \cup \text{GetAllSingleton}(OpenSSLContextImpl)$
$Objs \leftarrow Objs \cup \text{GetAllSingleton}(SSLParametersImpl)$
**for all** $o \in Objs$ **do**
    $Contexts \leftarrow Contexts \cup \text{GetClientContext}(o)$
**end for**
**for all** $Ctx \in Contexts$ **do**
    $OrgSize \leftarrow Ctx.\text{getSessionCacheSize}()$
    $Ctx.\text{setSessionCacheSize}(0)$
    $Ctx.\text{setSessionCacheSize}(OrgSize)$
**end for**
$ConfigAwareConnectionPool.\text{getInstance}().\text{clear}()$
System.gc()
```

Fig. 12: Clearing sessions by hooking Android Activity.

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

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

3) Concurrent eager deletion: Our last suggestion is to perform the eager deletion for out-dated `SSLSession` objects in the session cache. This patch uses a secondary thread, running in the background. The clean-up thread is created when the first `SSLSession` object is appended to the empty session cache. Once it is triggered, it looks for session cache entries which are expired and deletes them. It otherwise goes to sleep for a suitable amount of time (e.g., one minute), wakes up, and repeats the process. If the session cache is ever completely empty, the thread can terminate, and restart again as necessary.

We believe this is the most effective solution because it removes sessions in a timely manner. We find no long-lived master secrets after applying this patch. Furthermore, the overhead of this solution is fairly minimal, since the second thread spends most of its time asleep, and when it does its work, it's not correlated in time with any other user events, so the additional system impact is unlikely to be noticeable. Lastly, this patch requires a change only in the `AbstractSessionContext` class, so the patch is quite minimal to the Android codebase.

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

B. Observations and Future Work

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

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

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

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

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

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

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

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

C. Android 8

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

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

VII. Related Work

A. Memory Forensics

Memory forensics can be largely categorized into acquisition and analysis techniques. Regarding acquisition, Halderman et al. developed "cold-boot attack" [12] showing that an adversary can read out contents of memory, identifying encryption keys from their expanded key schedule; we used a similar technique to identify SSL session structures. Other researchers have looked at vulnerabilities in ARM's TrustZone [28], allowing a malicious app to obtain full system RAM.

In terms of analysis techniques, signature-based frameworks [35], [21] have been widely used. Various efforts have been made to identify structures by generating robust invariant [9], [16] and using static analysis [4].

A number of authors have looked at Android-specific issues in memory forensics. Sylve et al. [29] first proposed a technique for extracting physical memory from Android devices. Our research utilizes this technique for implementing our test framework. In 2013, Müller et al. [19] showed that cold-boot attacks are also applicable to Android phones.

Memory analysis on Android is commonly focused on extracting sensitive data from applications such as login IDs and passwords. Apostolopoulos et al. [2] showed that login credentials could be recovered from memory images using simple pattern matching. Hilgers et al. [14] identified a variety of data structures in memory images (e.g., GPS coordinates within photo metadata). Thing et al. [32] proposed an automated system that analyzes live memory on Android devices and showed it is possible to extract messages. DEC0DE [34] proposed a technique to extract plain-text call logs and address book entries from phone storage using probabilistic finite state machines. There have also been studies on specific texting applications such as WhatsApp [1], WeChat [37], and Viber [17]. One clever technique involves recovering previous GUI screens by piecing together the state of the Android widget view hierarchy [26], [27].

To the best of our knowledge, there is no in-depth study of cryptographic secrets of TLS on Android, but, for other platforms, there have recently been several studies for extracting TLS secrets from a virtual machine [31], Windows OS [15], and Oracle's Java HotSpot JVM [22]. The first two studies look at extracting master secrets; the latter focuses on general data reconstruction from a garbage-collected runtime system.

B. Android and TLS Security

A full consideration of prior work in Android security is beyond the scope of this paper, although Enck et al. [11] provide a nice survey paper. Reaves et al. [24] similarly summarize efforts to apply static and dynamic analysis to Android. Of note, they conclude that no existing tool is suitable for analyzing the cross-language issues that we observed between Java and C. Also, we limit our scope to Android framework and do not attempt to survey the millions of Android applications for how they use TLS, Egele et al. [10] looked at exactly this issue, identifying a large number of Android apps that misuse or misunderstand the correct use of cryptographic APIs.

C. Mitigation for memory disclosure attack

One of the main causes of data exposure is the insecure deletion which leads to leaking sensitive data [30]. Chow et al. addressed those problems in desktop and server with secure deallocation [5], [6].

One recent trend against memory disclosure attack is to maintain data outside of main memory. Tang et al. [30] suggested CleanOS that encrypts data with a secret key and evicts that key to a secure cloud storage. Also, `TRESOR` [18] proposed the register-based encryption technique without leaking information into memory. `Sentry` [7] was developed to maintain data in the cache or internal memory in SoC chip. `CaSE` [36] was proposed to keep sensitive data using TrustZone from both physical and software-based memory disclosure attacks.

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

VIII. Conclusion

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

As of the camera-ready deadline for this paper, we have reported the issues and mitigations described here to Google through their standard security vulnerability reporting process. We have not yet received any word on their plans to address these issues.

Acknowledgements

We thank our anonymous reviews and our shepherd for their assistance and suggestions. We also thank Adam Pridgen, for providing the infrastructure upon which we built our own analyses. This work was supported in part by NSF grants CNS-1409401 and CNS-1314492.

References

[1] C. Anglano, "Forensic analysis of WhatsApp Messenger on Android smartphones," *Digital Investigation*, vol. 11, no. 3, pp. 201–213, 2014.

[2] 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.

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

[4] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *CCS '09*, 2009.

[5] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *13th USENIX Security Symposium*, 2004.

[6] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," in *14th USENIX Security Symposium*, 2005.

[7] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting data on smartphones and tablets from memory attacks," in *ASPLOS '15*, Istanbul, Turkey, 2015.

[8] T. Dierks and E. Rescorla, *RFC5246: The transport layer security (TLS) protocol, version 1.2*, IETF, Aug. 2008, https://tools.ietf.org/html/rfc5246.

[9] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *CCS '09*, 2009.

[10] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *CCS '13*, 2013.

[11] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android security," *IEEE security & privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[12] 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*, 2008.

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

[14] 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.

[15] J. Kambic, "Cunning with CNG: Soliciting secrets from Schannel," in *Black Hat USA*, 2016.

[16] 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 *NDSS '11*, 2011.

[17] A. Mahajan, M. Dahiya, and H. Sanghvi, "Forensic analysis of instant messenger applications on Android devices," *International Journal of Computer Applications*, no. 8, pp. 38–44, 2013.

[18] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *20th USENIX Security Symposium*, 2011.

[19] T. Müller and M. Spreitzenbarth, "FROST: Forensic recovery of scrambled telephones," in *International Conference on Applied Cryptography and Network Security*, 2013.

[20] C. Ntantogian, D. Apostolopoulos, G. Marinakis, and C. Xenakis, "Evaluating the privacy of Android mobile applications under forensic analysis," *Computers & Security*, vol. 42, pp. 66–76, 2014.

[21] 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.

[22] 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*, 2017.

[23] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill, "Studying TLS usage in Android apps," in *Proceedings of the 13th ACM Conference on Emerging Networking Experiments and Technologies*, 2017.

[24] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife *et al.*, "* droid: Assessment and evaluation of Android application analysis tools," *ACM Computing Surveys*, vol. 49, no. 3, p. 55, 2016.

[25] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig, *RFC5077: Transport layer security (TLS) session resumption without server-side state*, IETF, Jan. 2008, https://tools.ietf.org/html/rfc5077.

[26] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "GUITAR: Piecing together Android app GUIs from memory images," in *CCS '15*, 2015.

[27] 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*, 2016.

[28] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "TrustDump: Reliable memory acquisition on smartphones," in *European Symposium on Research in Computer Security*, 2014.

[29] 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.

[30] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "CleanOS: Limiting mobile data exposure with idle eviction," in *OSDI '12*, 2012.

[31] B. Taubmann, C. Frädrich, D. Dusold, and H. P. Reiser, "TLSkex: Harnessing virtual machine introspection for decrypting TLS communication," in *Proceedings of DFRWS EU Annual Conference*, 2016.

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

[33] M. Vanhoef and F. Piessens, "Key reinstallation attacks: Forcing nonce reuse in WPA2," in *CCS '17*, 2017.

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

[35] A. Walters, "The volatility framework: Volatile memory artifact extraction utility framework," 2007.

[36] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-assisted secure execution on ARM processors," in *IEEE Symposium on Security and Privacy (SP)*, 2016.

[37] F. Zhou, Y. Yang, Z. Ding, and G. Sun, "Dump and analysis of Android volatile memory on WeChat," in *IEEE International Conference on Communications*, 2015.