# Enabling the forensic study of application-level encrypted data in Android via a Frida-based decryption framework

# Enabling the forensic study of application-level encrypted data in Android via a Frida-based decryption framework

Cosimo Anglano
cosimo.anglano@uniupo.it
University of Piemonte Orientale
Alessandria, Italy
Consorzio Interuniversitario
Nazionale per le Telecomunicazioni
Parma, Italy

Massimo Canonico
massimo.canonico@uniupo.it
University of Piemonte Orientale
Alessandria, Italy
Consorzio Interuniversitario
Nazionale per le Telecomunicazioni
Parma, Italy

Andrea Cepollina
20024305@studenti.uniupo.it
University of Piemonte Orientale
Alessandria, Italy

Davide Freggiaro
20024629@studenti.uniupo.it
University of Piemonte Orientale
Alessandria, Italy

Alderico Gallo
20024636@studenti.uniupo.it
University of Piemonte Orientale
Alessandria, Italy

Marco Guazzone
marco.guazzone@uniupo.it
University of Piemonte Orientale
Alessandria, Italy
Consorzio Interuniversitario
Nazionale per le Telecomunicazioni
Parma, Italy

## Abstract

The forensic study of mobile apps that use application-level encryption requires the decryption of the data they generate. Such a decryption requires the knowledge of the encryption algorithm and key. Determining them requires, however, a quite complex analysis that is time-consuming, error prone, and often beyond the reach of many forensic examiners. In this paper, we tackle this problem by devising a framework able to automate the decryption of these data when third-party encryption libraries or platforms are used. Our framework is based on the use of dynamic instrumentation of app's binary code by means of *hooking*, which enables it to export the plaintext of data after they have been decrypted by the app, as well as the corresponding encryption key and parameters. This framework has been conceived to be used only with test devices used for forensic study purposes, and not with devices that need to be forensically analyzed. We describe the architecture of the framework as well as the implementation of its components and of the hooks supporting three prominent and popular encryption libraries, namely `SQLCipher`, `Realm` and `Jetpack Security`. Also, we validate our framework by comparing its decryption results against those published in the literature for `Wickr Me`, `Signal`, `Threema`, and `Element`.

**ACM Reference Format:**
Cosimo Anglano, Massimo Canonico, Andrea Cepollina, Davide Freggiaro, Alderico Gallo, and Marco Guazzone. 2023. Enabling the forensic study of application-level encrypted data in Android via a Frida-based decryption framework. In *The 18th International Conference on Availability, Reliability and Security (ARES 2023), August 29-September 1, 2023, Benevento, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3600160.3605029

## 1 Introduction

The forensic study of mobile applications (henceforth referred to as *apps* for brevity) is currently one of the prominent research areas in the mobile forensics field. The goal of such a study is the characterization of the behavior of a specific app in terms of the relationship between the actions performed by a user and the data it generates as consequence of those actions. As a matter of fact, if these relationships are known to an examiner, (s)he may infer the possible occurrence of a specific user action if the data generated by that action are found on the device.

To make these relationships explicit, the app must be studied in order to determine (a) which data it generates in response to the various user actions, (b) where it stores these data, and (c) how it encodes them [5–7, 17, 20, 31]. This study is typically performed by exercising, in a systematic manner and in a controlled environment, the functionalities provided by the app in order to elicit the generation of data that, consequently, can be associated with the corresponding user action [8].

The above experimental study is based on the assumption that the data generated by an app are available to the experimenter. However, application-level encryption, whereby apps encrypt the data they generate before storing them on the device, is making this difficult. As a matter of fact, these (encrypted) data need to be decrypted first (i.e., they must be available in the so-called *plaintext*), an operation that requires the experimenter to know both the encryption algorithm and the encryption key that have been used by the app.

This raises the following two challenging issues for the experimenter:

(1) the encryption key is never exposed by an app (and this occurs also frequently with the encryption algorithm), so

these information need to be discovered by means of suitable analysis techniques, which are however quite complex to carry out;

(2) the data modified by the app needs to be inspected continuously, ideally after each action carried out on the app during the experimentation. This induces a loop of modify–extract from device memory–decrypt the data, which significantly increases the time and effort of carrying out the forensic study.

In this paper, we propose a framework able to address both the issues mentioned before in scenarios where encryption is achieved by using third-party software components, as discussed below.

## 1.1 Problem definition

Generally speaking, there are two possible ways for an app to implement encryption. The first one involves using homemade encryption code. This approach, however, is more and more deprecated, as it is quite difficult to write correct and robust encryption code [14]. The second approach, which is adopted by the vast majority of modern mobile apps, is to use third-party encryption code, which is provided either as a library of functions (e.g., `Android JetPack Security` [3] or `javax.crypto` [2]), or as an encryption service running locally (e.g., encrypted databases like the `SQLCipher` [37] extension to SQLite or the `Realm` [35] DBMS). This latter approach is more robust and secure than homemade one. In this paper, we thus focus on apps using the latter approach, since it allows us to maximize the applicability of the framework we have developed.

Third-party encryption code provides two distinct methods to encrypt application data, which can be used either individually or in combination, namely [27]: (a) encrypt data by prior to their storage into a file or a database; (b) store plaintext data into an encrypted database.

In both cases, to obtain the plaintext of these data, the experimenter needs to determine the encryption algorithm, parameters, and key used by the app, which may entail performing a quite complex reverse engineering of that app and of the encryption code. More specifically, reverse engineering of these apps entails the combined use of both static and dynamic analysis techniques, and requires both technical skills which are typically not in the possession of all experimenters, and a significant amount of manual work [16, 18, 19, 24].

It is important to note that application-level encryption adds an extra layer of encryption to the other encryption layers used by modern mobile operating system, such as *File-Based Encryption* [25] (FBE). As a consequence, even after the operating system decrypts its file systems when the experimenter unlocks the device, data encrypted by an app remains unencrypted.

## 1.2 Our framework

The framework we propose in this paper is able to automate the extraction of plaintext data from the memory space of a running app. As a matter of fact, encrypted data stored on persistent storage needs to be necessarily decrypted before being load in main memory when the app needs to process it (in other words, these data are encrypted only when they are stored on persistent storage). The corresponding plaintext can be therefore obtained by intercepting at run time the calls made by the app code to all the functions that decrypt data, and by modifying the behavior of these functions in order to force them to export these data to external storage for the perusal of the experimenter.

Function call interception is achieved through a technique known as *hooking* [13], which consists in dynamically instrumenting the binary code of the app so that (a) the calls to specific functions are intercepted at run-time, and (b) external code injected into the application space (the *hook*) is executed before the actual execution of the function code.

This process is conceptually depicted in Fig. 1, where we show five apps: `App 1` and `App 2` use `Data encryption library X` to encrypt their data, while instead `App 3, App 4`, and `App 5` use `Database encryption library Y`.

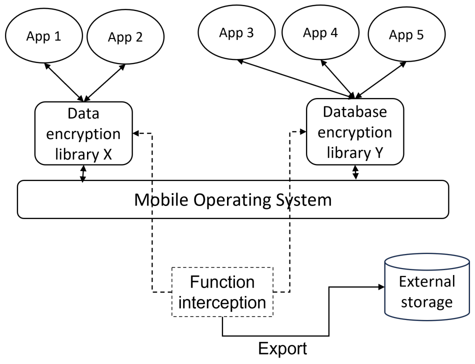

**Figure 1: Conceptual approach.**

As shown in Fig. 1, in our approach function calls interception is carried out at the level of the individual encryption software components, e.g. `Data encryption library X` and `Database encryption library Y`. This is achieved by injecting, into the app's binary code, a hook specific to the encryption library it uses.

An important point to make here is the fact that hooks work at the function level, so once the hooks for a given encryption library have been developed, it can be used to get the plaintext out of any app using that library. For instance, in Fig. 1, plaintext data for `App 1` and `App 2` is extracted using the hooks for `Data encryption library X`, while for `App 3, App 4`, and `App 5` the hooks for `Database encryption library Y` are used.

Our framework exploits this fact to provide automation in plaintext extraction. In particular, it automates the injection of hooks into an application under study, provided that the hooks for the encryption library it uses are available, and the extraction and storage of plaintext data. To the best of our knowledge, there is no other similar framework that has been published in the literature.

In its current implementation, our framework supports two prominent and widely-used encryption libraries, namely `SQLCipher` [37] for SQLite databases and `Jetpack Security` [3] for files (support for the latter is only partial, but work is ongoing to complete it), and the `Realm` [34], a DBMS for mobile apps which is an alternative to SQLite and provides encryption as a built-in feature.

To carry out hooking, our framework relies on the `Frida` toolkit [28] for binary instrumentation, so it can be implemented on any operating system supported by `Frida`, and in particular on both

Android and iOS. In this paper, however, we focus on Android, and we leave the iOS case to future work.

As a final consideration, it should be noted that our framework requires that the apps under study are executed with super-user privileges (i.e, on rooted Android devices). This, however, is not a real limitation since, as discussed before, the focus of our work is on the experimental forensic study of mobile apps, which entails the use of test devices under the complete control of the experimenter, and the interaction with the app to elicit the generation of data. *In other words, our framework is not intended to be used for the forensic analysis of app data stored on the seized devices.*

## 2 Related work

The problem of decrypting the data encrypted by apps, both for Android and other operating systems, has been already tackled in the literature.

[15] studies the decryption of databases of the `Telegram X` and `BBM-enteprises` applications, both for Android and for Windows platforms. They found out that both applications use the `SQLCipher` encryption library, and perform manual reverse engineering to discover the encryption parameters and the key generation procedure used by them. [18] focuses on the decryption of databases of the `NateOn`, `KakaoTalk`, and `QQ` messengers on Windows platforms. They also perform manual reverse engineering to discover the key generation and the encryption procedure used by these apps, all of which use home-brewed encryption code. In [24], authors analyze 56 note and journal Android apps, and find out that, while 95% store their data locally in an insecure way, the remaining 5% use home-brewed encryption libraries for their databases, and perform manual reverse engineering to recover the key generation procedure and the encryption algorithms and parameters. [19] focuses on the decryption of the databases for the `Signal`, `Wickr`, and `Threema` instant messengers. Authors find that all these apps rely on the `SQLCipher` library, and a thorough manual reverse engineering is carried out in order to determine the key generation process and the encryption parameters. [27] proposes an approach, based on static code analysis, that automates the process of discovering whether an Android app encrypts its database, and in some cases also the encryption method that is used. However, no means of recovering the encryption key and parameters for the analyzed apps is provided.

All these works show the relevance of the problem we tackle with our framework, and that the manual recovery of the encryption key and the other encryption parameters, required to decrypt databases, is a complex, time-consuming, and possibly error-prone process. In contrast, our framework, by automating the extraction of these parameters from prominent database encryption libraries, avoids the problems characterizing manual analysis.

Other works have instead focused on the decryption of backups generated by utilities developed by smartphone vendors, such as [21–23]. These works focus on a problem different from that targeted by this paper, so they do not compare to our work directly.

There are other works in the literature that rely on hooking to obtain relevant information from the memory space of running processes. [26] proposes a hooking framework aimed at monitoring sensitive methods in shared objects, while [13] proposes a framework for the automated analysis of app code with the aim of identifying hooks placed by malicious code. These works, however, deal with problems different from the one we address in this paper.

## 3 The decryption framework

As we already mentioned, our framework works by dynamically (i.e., at run time) instrumenting the app binary code to hook the decryption functions it calls, so that the calls to these functions can be intercepted and, consequently, the plaintext data they obtain after decryption can be exported outside the app. Furthermore, if the encryption key and parameters are passed to these functions, our framework exports them too.

In this section, we present the above framework by first discussing its architecture (Sec. 3.1), then by illustrating some guidelines on (a) how it can be concretely implemented (Sec. 3.2) and on (b) how it can be used with a specific app (Sec. 3.3).

### 3.1 Framework architecture

As we anticipated in Sec. 1, our framework is based on `Frida`, an open source, multi-platform, binary dynamic instrumentation toolkit. `Frida` is able to inject a custom script (written either in the `JavaScript` or the `TypeScript` languages) into a running process, and to hook that script to any function called by the process. In this way, when a hooked function is called by the running process, the corresponding script is executed either prior to or after the execution of the function, depending on which sequence is required to get plaintext data.

The architecture of our decryption framework is schematically depicted in Fig. 2, where the components we developed are highlighted in light gray, while the ones shown in white are those belonging to the `Frida` toolkit.

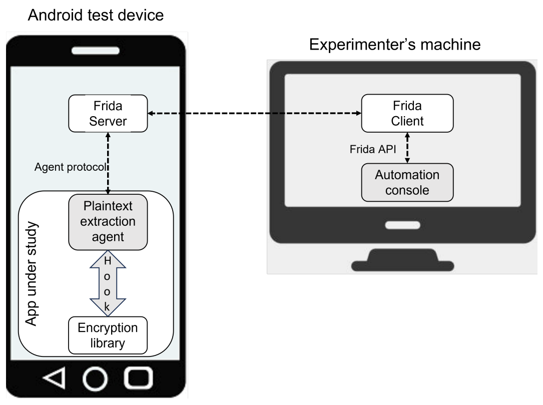

**Figure 2: Framework architecture.**

As shown in the figure, the framework runs on two distinct systems: (a) an Android device, where the app of interest is running, and (b) an experimenter's machine, to which the device is connected. It encompasses four distinct components, two of which belong to `Frida`, while the other three ones have been developed by us. In particular:

- the `Plaintext extraction agent`: it is the code, injected into the app binary code, which takes care of accessing the plaintext data and of exporting them to external storage. The agent includes also a hook, which is the code injected into the app binary code that starts the execution of the agent when the functions of the `Encryption library` that decrypt data are called by the app;

- the `Automation console`: it provides the experimenter with suitable mechanisms allowing to inject specific agents and hooks for the app under study;
- `Frida` toolkit components: they are used to inject hooks and agents into the running app, and to collect the results generated by the agents above. More specifically, they are:
  – the `FRIDA Server`: it spawns the process of the app under study, injects the `Plaintext extraction agent` into this process, hooks it to the encryption/decryption functions, and captures the output generated by it. The `FRIDA Server` needs super-user privileges to run properly so, as already anticipated, our framework assumes that the Android device is *rooted*; however, this is not a real restriction, as devices used in the forensic study of apps are typically under the complete control of the experimenter; [1]
  – the `FRIDA Client`: it runs on the experimenter's machine, and interacts with both the `Automation console` and the `FRIDA Server`. In particular, it accepts user commands, forwards them to the server, waits for the server to complete the command, receives the output from the server, and forwards it to the user.

### 3.2 Implementing the framework for a specific encryption library

To support a given encryption library, the only framework component that needs to be implemented for each specific encryption library is the `Plaintext extraction agent`. The other components are instead library-independent and, therefore, they do not need to be modified when support for a new library is developed.

In this section, we illustrate the generic procedure that can be followed to carry out this implementation (we will illustrate how this generic procedure can be instantiated in practice in Sec. 4).

To implement the `Plaintext extraction agent` for a specific library, the first step to be carried out is the identification of the functions of that library that need to be hooked, by following the guidelines discussed below (where we use the generic term *container* to denote both a file and a database):

- *obtaining plaintext data*: the plaintext of all the data that have been saved into an encrypted container while an app is running can be obtained by intercepting the functions that "close" that container (i.e., those functions that are called when the app no longer needs to use those data), and by reading its contents before passing the control to the function that will actually close it. Sometimes, it is necessary to intercept also the functions that "open" the container if the encryption library does not provide "close" functions (e.g., see Sec. 4.3 as an example).
- *obtaining the encryption key and parameters*: these information are typically (though not always) exposed by the app if the functions used to "open" the container prior to its use requires them. In case they are exposed, the encryption key and parameters can be obtained by intercepting the calls to these functions, and having the hook code access and print the values of the function parameters corresponding to the above key and parameters.

These functions are typically identified by examining the documentation of the encryption library and/or its source code. In case neither the source code of the library, nor its documentation are available, then a reverse engineering phase of the library needs to be performed.

After the functions to be hooked have been identified, the corresponding `Plaintext extraction agent` is developed by writing a set of hooks for them, as well as the code that is executed by the hook when the corresponding function is called. The implementations of both parts depend on whether the library has been written in Java or as *native code* (i.e., using the Android NDK toolset [29]). In the remainder of this section we discuss how to associate a hook with the corresponding function, while the implementation of the code executed by the hook – which depends on the specific encryption library it targets – is discussed in Sec. 4 for three distinct widely-used libraries.

**Encryption library written in Java.** To attach a hook to a generic Java method `class_to_hook`, which corresponds to the generic `<x>.<y>.<class_to_hook>` fully qualified class name (e.g., `io.realm.RealmConfiguration`), the code skeleton shown in Fig. 3 can be used. As shown in Fig. 3, the `FRIDA Java.perform(fn)` function is

```
1  Java.perform(function x() {
2      const <class_to_hook> = Java.use("<x>.<y>.<class_to_hook>")
3      <class_to_hook>.$init.overload(arg1,arg2,...,argN)
4      .implementation = function (val1,val2,...,valN) {
5      <hook_code>
6      const toRet = this.$init(val1,val2,...,valN);
7      return toRet
8      };
9  });
```

**Figure 3: Attaching a hook to a `Java` method.**

used (line 1) to attach the hook to the chosen Java method. Within this function, we first specify the name of the class to be hooked, and its fully qualified method name (line 2). Then, we specify which method of that class we want to hook to (line 3), which in Fig. 3 is the constructor (`$init`) of class `class_to_hook`, by also specifying its signature (i.e., the list of its arguments).

Next, we specify the code of the hook using the `.implementation` keyword (line 4). As indicated in Fig. 3, we can insert any code (generically denoted as `<hook_code>`) either before (as indicated in the figure) or after the call to the original function (line 6).

**Encryption library written in native code.** To attach a hook to a native library function named, e.g., `LIB`, the code of the `Plaintext extraction agent` (expressed in `TypeScript`) – which is shown in Fig. 4 – is more complex than the one used with a Java library. To hook a specific native library `LIB`, we must be sure that `LIB` has been already loaded into main memory by the app. However, in an Android app, native libraries might not be loaded in memory

[1]We made this decision in order to make the framework app independent, i.e. able to run with any app without having to customize it for that specific application. The alternative was indeed to run the `FRIDA Server` without super-user privileges, but this required to modify the executable code of each app in order to include into it a special-purpose `FRIDA` library, that gets loaded when the app is started.

```
1  let libraryLoaded = false
2  Java.performNow(function x() {
3    const System = Java.use('java.lang.System');
4    const Runtime = Java.use('java.lang.Runtime');
5    const VMStack = Java.use('dalvik.system.VMStack');
6    function loadedHookedLibrary(library: string) {
7      if ((library === 'LIB' || library.endsWith('LIB.so')) && !libraryLoaded) {
8        // Make sure to hook the LIB library just once
9        loadNativeHooks() // Load hooks for the LIB library
10       libraryLoaded = true
11     }
12   }
13
14   // Hook for the System.loadLibrary() method
15   System.loadLibrary.implementation = function (libName: string) {
16     try {
17       // Load the given library as expected by the app and then perform
          function hooking
18       const loaded =
          Runtime.getRuntime().loadLibrary0(VMStack.getCallingClassLoader(),
          libName);
19       loadedHookedLibrary(libName)
20       return loaded
21     } catch (ex) {
22       // Code for exception handling (not shown for brevity)
23     }
24   };
25
26   // Hook for the System.load() method
27   System.load.implementation = function (libFile: string) {
28   try {
29       // Load the given library as expected by the app and then perform
          function hooking
30       const loaded = Runtime.getRuntime().load0(VMStack.getStackClass1(),
          libFile);
31       loadedHookedLibrary(libFile)
32       return loaded
33     } catch (ex) {
34       // Code for exception handling (not shown for brevity)
35     }
36   };
37 })
```

**Figure 4: Attaching a hook to the `System.loadLibrary()` and `System.load()` methods.**

when the app is started, but later on demand by explicitly invoking suitable Java methods, that is `System.loadLibrary` and `System.load`. Hence, we must hook the above two methods so that, when they are called by the app, they associate the hook with the target library `LIB`. Such a hooking is actually performed by function `loadedHookLibrary`, which is defined in lines 6– 12, and is called by both `System.loadLibrary` (line 19) and `System.load` (line 31).

These actions need to be performed as soon as possible to prevent that `System.load` or `System.loadLibrary` are invoked by the app before hooks have been installed, thus missing the calls to them. This is achieved by using the `Java.performNow(fn)` `FRIDA` method (see line 2 of Fig. 4), which is executed by `FRIDA` just after the Java Virtual Machine has started but before any app-specific class is loaded.

In particular, we hook the `loadLibrary` and `load` methods of both the `System` and the `Runtime` classes.[2] The hooking of the `System.loadLibrary` method is achieved in lines 15 – 24, while that of the `System.load()` method is achieved in lines 27 – 36. As can be seen, in both cases when the method is loaded, the `loadedHookedLibrary` (lines 6 – 12) is called (lines 19 and 31, respectively). Specifically, the hooking of the `System.loadLibray` method consists in (1) loading the library `libName` (as expected by the invoking application) through the Java `Runtime.loadLibrary0` method (line 18) and then (2) installing the hooks for the interested functions (line 9). Similarly, the hooking of the `System.load` method performs the same steps but it uses the Java `Runtime.load0` method to load the library stored in the file `libFile`.

[2]In Fig. 4, because of space constraints and to avoid cluttering, we omit the hooking of the `Runtime` methods, which however is similar to that of the `System` methods.

## 3.3 Using the framework with an app

After a `Plaintext extraction agent` for a specific encryption library has been developed, to use it with a specific app using that encryption library requires to determine whether the app uses that library or not. The ways in which this can be determined differ according to the availability or not of the app source code.

If source code is available, then it can be inspected to determine which library it uses. In some cases, the source code may also include a *build file* that explicitly lists the libraries from whom the app depends, and that of course includes also the reference to the encryption library. As an example, the excerpt in Fig. 5, extracted from the source code of the `Element` [11] secure messaging app (which it is known to use encrypted databases), shows the contents of its *build.gradle* file [12].

```
1 apply plugin: 'com.android.library'
2 apply plugin: 'kotlin-android'
3 apply plugin: 'kotlin-kapt'
4 apply plugin: 'kotlin-parcelize'
5 apply plugin: "org.jetbrains.dokka"
6 apply plugin: 'realm-android'
7 //[...]
```

**Figure 5: Excerpt from the `Element` app source code `build.gradle`.**

As can be seen from line 6, the `Element` app uses the `Realm` library to encrypt its databases.

If the application is, instead, closed-source, a different approach needs to be used. In particular, its *APK* file (i.e., the package file which is installed on the device) needs to be obtained first, and then unpacked and de-compiled using a tool like `Apktool` [30]. Once these steps have been carried out, it is possible to inspect the resulting code to look for the names of the included library files, among which there will be also the encryption library. Typically, the code of the app includes a folder, named after the so-called *reverse domain name notation* of the library. For instance, in the case of the `SQLCipher` library, this folder is named `net.zetetic.database.sqlcipher`.

Once the encryption library used by the app has been determined, and the availability of the corresponding `Plaintext extraction agent` has been ascertained, then such agent is hooked to the app while it is running.

As mentioned before, this step is carried out by the `FRIDA Client`, which requires that the user specifies the names of the app (more precisely, its *package name*, e.g. `com.whatsapp`) and of the file storing the code of the `Plaintext extraction agent` to attach, and sends to the `FRIDA Server` the commands that make it hook the agent with the chosen app.

To illustrate how this is done in practice, let us discuss the excerpt of the `FRIDA Client` (written in Python) shown in Fig. 6. The first action which is carried out is the spawn of the app on the device

```
1 import frida
2 #[...]
3 pid = frida.get_usb_device().spawn(package)
4 session = frida.get_usb_device().attach(pid)
5 script = session.create_script(agent.read())
6 script.load()
7 frida.get_usb_device().resume(pid)
```

**Figure 6: An excerpt of the `FRIDA Client`.**

(line 3), which is followed by the creation of a *FRIDA session* whereby the `FRIDA Server` attaches itself (i.e., controls the execution) to the spawned app process (line 4). Then, the `Plaintext extraction agent` code is read and associated with that session (line 5), and subsequently injected into the code of the app (line 6). Finally, the execution of the app is resumed (line 7).

Now, when the app calls the functions hooked by the `Plaintext extraction agent`, its code is executed and the corresponding action is carried out.

## 4 Use cases

In this section, we discuss the design and implementation of the `Plaintext extraction agent` for three prominent and widely-used encryption libraries, namely `SQLCipher` and `Realm` for database encryption, and `JetPack Security` for file encryption.

The purpose of this section is two-fold. On the one hand, we demonstrate the wide applicability of our framework to a large set of apps, since these three libraries are used by a very large set of apps, and their usage will likely grow in the future. On the other hand, we describe techniques that can be used to implement agents for other encryption libraries, thus widening the base of potential users of our framework.

### 4.1 Decrypting `SQLCipher` databases

`SQLCipher` currently is the de-facto standard for database encryption in Android apps, and it works in conjunction with `SQLite` [9] (which is in turn the de-facto standard for database support for mobile apps). Therefore, it was a natural choice to include support for its decryption in our framework.

`SQLCipher` is a native library (written in C), although can be accessed by an Android app either directly using the `SQLCipher` *for Android* classes [38] or through the Android's *Room* framework [1]. Therefore, to deal with all possible `SQLCipher` integration scenarios, it is sufficient to dynamically instrument the native library.

In practice, this corresponds to placing the hooks discussed below in the `loadNativeHooks` function shown in Fig. 4, where the `LIB` placeholder has been replaced with the `sqlcipher` word (see Fig. 4, lines 7–9).

Let us now describe the design and the implementation of the `SQLCipher Plaintext extraction agent` (in the following `SQLCipher Agent` for brevity), by discussing its handling of multiple open databases by the app under study (Sec. 4.1.1), the extraction of plaintext database contents (Sec. 4.1.2) and of the encryption key and parameters (Sec. 4.1.3).

**4.1.1 Handling multiple open databases** A peculiar feature of `SQLite` is that it allows the same app to open and use multiple distinct databases during its operations. This means that the `SQLCipher Agent`, to work correctly, needs to be able to associate both the plaintext data and the encryption key and parameters with the right database, in case several of them are simultaneously used by the app,

We achieve this goal by having the agent hook the `sqlite3_open` function (and its variants) to store the database handle returned by such function, and using it later when extracting the plaintext or the encryption key and parameters associated with that database. The `TypeScript` hook code is reported in Fig. 7, where we see that

```
1  class NativeDb { // Store information about an open database
2    handle: string // The database handle
3    key_hex: string | null // The encryption key as hex-string
4    ...
5  }
6  let dbDict = new Map<string, NativeDb>();
7  const sqlcipher = Process.getModuleByName("libsqlcipher.so");
8  Interceptor.attach(sqlcipher.getExportByName("sqlite3_open"), {
9    onEnter: function (args) {
10     this.filename = args[0].readUtf8String(); // Database filename
11     this.ppDb = args[1]; // On exit, database handle
12   },
13   onLeave: function (retval) {
14     const dbHandle = this.ppDb.readPointer()
15     // Store the database handle for later use
16     dbHandleStr = String(dbHandle)
17     db = NativeDb(dbHandleStr)
18     db.path = this.filename
19     dbDict.set(dbHandleStr, db)
20   },
21 });
```

**Figure 7: `SQLCipher`: hooking of the `sqlite3_open` function to extract the database handle.**

the database handle, as returned by the original `sqlite3_open`, is first extracted (line 14) and then stored in an in-memory dictionary `dbDict` for later use (lines 16–19). [3]

**4.1.2 Extracting plaintext contents** As mentioned in Sec. 3.1, to obtain plaintext data we need to hook the function that closes the database, which in the `SQLCipher` case is the `sqlite3_close` function (and its variants).

This hooking is performed by using the `Frida`'s `Interceptor` API, which allows to attach up to two callbacks to a hooked function, namely `onEnter` (invoked just before executing the original code of the hooked function) and `onLeave` (invoked just before returning from the hooked function), and takes care of executing the original code of the hooked function (among the invocation of the above callbacks) so as to preserve its intended behavior.

For the extraction of plaintext data from the database, it suffices to use the `onEnter` callback, as shown in Fig. 8, where the function `dumpDbToPlainText` is executed each time the `sqlite3_close` function is called by the app (line 5). In particular, the above function invokes the `sqlite3_exec` function (line 13) to query the database (represented by the `NativeDb` object passed as input argument) and save the results (i.e., the plaintext contents of the database) to a text file in the same folder where the encrypted database file is stored.

**4.1.3 Extracting encryption key and parameters** `SQLCipher` provides several security specific parameters (which are listed in Table 1) to control the encryption of a database. Among them, the only mandatory parameter required to encrypt a database is the

[3]The hooks for the other variants of `sqlite3_open()` function are very similar, and consequently are not shown here to avoid repetitions.

```
 1 Interceptor.attach(sqlcipher.getExportByName("sqlite3_close"), {
 2   onEnter: function (args) {
 3     this.dbHandle = args[0]; // Database handle
 4     const db = dbDict.get(this.dbHandle.toString())
 5     dumpDbToPlainText(db)
 6   }
 7 });
 8 const callable_sqlite3_exec = new
       NativeFunction(sqlcipher.getExportByName("sqlite3_exec"), 'int',
       ['pointer', 'pointer', 'pointer', 'pointer', 'pointer']);
 9 function dumpDbToPlainText(nativeDb: NativeDb) {
10   let pathPlaintext = nativeDb.path + ".native_plaintext"
11   const errorMsgPtr = Memory.alloc(Process.pointerSize)
12   const sqlQueryPtr = Memory.allocUtf8String("ATTACH DATABASE '" +
       pathPlaintext + "' AS plaintext KEY '';SELECT
       sqlcipher_export('plaintext');DETACH DATABASE plaintext;")
13   const retExec = callable_sqlite3_exec(ptr(nativeDb.handle), sqlQueryPtr,
       NULL, NULL, errorMsgPtr)
14 }
```

**Figure 8: `SQLCipher`: hooking of the `sqlite3_close` function to extract plaintext data from the database.**

| Name | Meaning |
|---|---|
| `key` | the encryption key |
| `cipher_kdf_algorithm` | key derivation function to be used |
| `kdf_iter` | number of iterations used with the key derivation function |
| `cipher_page_size` | page size for the encrypted database |
| `cipher_plaintext_header_size` | size of the header of the encrypted database that must not be encrypted |
| `cipher_use_hmac` | either enables or disables the use of a per-page HMAC |
| `cipher_hmac_algorithm` | the HMAC algorithm to be used |

**Table 1: Parameters used by `SQLCipher` to control encryption.**

encryption key `key`; the other parameters are optional and, if not explicitly set, suitable default values will be used for them. However, to successfully decrypt a database, all the above parameters must be set to the same values used at encryption time. Hence, all of them need to be extracted by the `SQLCipher Agent` in order to enable the offline decryption of a database.

The procedure to extract the above parameters is different for the encryption key and the encryption parameters.

To extract the encryption key of an open database, it is sufficient to hook the `sqlite3_key` and `sqlite3_rekey` functions (and their variants), as shown in Fig. 9. As shown in the figure,

```
 1 Interceptor.attach(sqlcipher.getExportByName("sqlite3_key"), {
 2   onEnter: function (args) {
 3     this.dbHandle = args[0]; // Database to be keyed
 4     this.key_ptr = args[1]; // The key
 5     this.key_size = args[2].toInt32(); // The length of the key (in bytes)
 6   },
 7   onLeave: function (retval) {
 8     if (retval.toInt32() == 0) {
 9       // The key has been successfully set in the database
10       let key_bytes = this.key_ptr.readByteArray(this.key_size);
11       const db = dbDict.get(String(this.dbHandle.toString()))
12       db.key_hex = arrayBufferToHexString(key_bytes)
13       Report_key_to_FRIDA_Client(db.key_hex)
14     }
15   },
16 });
```

**Figure 9: `SQLCipher`: hooking of the `sqlite3_key` function to extract the encryption key.**

the `SQLCipher Agent` extracts the values of the input arguments when entering the hooked function (including the encryption key; lines 3–5) and stores the extracted information in the in-memory dictionary `dbDict` before leaving that function, once it is sure that the key has been correctly set (lines 10–12). [4] Finally, the key is reported to the `FRIDA Client` (action denoted by the generic function `Report_key_to_FRIDA_Client` (line 7)).

Conversely, to extract the encryption parameters, we extend the hook to the `sqlite3_close` function (already shown in Fig. 8) to include also the queries for these values to the database engine. By doing so, we avoid the need of instrumenting all the possible `SQLCipher` functions that could change these values at run time. The extended code for the hook to the `sqlite3_close` function is shown in Fig. 10, where the unchanged parts of the code from Fig. 8 are replaced by comments in square brackets (lines 3 and 12).

```
 1 Interceptor.attach(sqlcipher.getExportByName("sqlite3_close"), {
 2   onEnter: function (args) {
 3     [CODE FROM LINE 3 TO LINE 5 OF Fig. 8 GOES HERE]
 4     // Note: pragmaKeys is a list of the PRAGMA names associated with the
         encryption parameters
 5     for (const pragmaKey of pragmaKeys) {
 6       db.params[pragmaKey] = getPragmaValue(db, pragmaKey) // Extract parameter
         pragmaKey
 7       Report_params_to_FRIDA_Client(db.params[pragmaKey])
 8     }
 9
10   }
11 });
12 [CODE FROM LINE 8 TO LINE 14 OF Fig. 8 GOES HERE]
13 function getPragmaValue(nativeDb: NativeDb, name: string) {
14   // Runs a PRAGMA statement to get the value associated with 'name'
15   const errorMsgPtr = Memory.alloc(Process.pointerSize)
16   let val: string[] = [];
17   const callback = new NativeCallback((_arg1, count, data, columns) => {
18     for (let i = 0; i < count; i++) {
19       const arrayElementPointer = data.add(Process.pointerSize *
         i).readPointer()
20       const value: string | null = arrayElementPointer.readUtf8String()
21       if (value !== null)
22         val.push(value);
23     }
24     return 0;
25   }, 'int', ['pointer', 'int', 'pointer', 'pointer']);
26   const retExec = callable_sqlite3_exec(ptr(nativeDb.handle),
       Memory.allocUtf8String("PRAGMA " + name + ";"), callback, NULL,
       errorMsgPtr)
27   return val;
28 }
```

**Figure 10: `SQLCipher`: extension to the hook for `sqlite3_close` to extract the encryption parameters.**

As shown in Fig. 10, the encryption parameters are obtained by executing a series of SQL PRAGMA statements (embedded into the `getPragmaValue` function (lines 13 – 28)) to retrieve the value of the encryption parameters just before the database is closed. This function is repeatedly called by the `onEnter` callback (see line 5); the result of each call is stored in the in-memory dictionary `db` (line 6). At the end of this sequence of calls, the results are reported to the `FRIDA Client` (action denoted by the generic function `Report_params_to_FRIDA_Client` (line 7)).

## 4.2 Decrypting `Realm` databases

`Realm` is a database library (available for both Android and iOS) which provides support to create and manage object-oriented database for mobile applications, and whose adoption by apps is rapidly

[4] The hooks for the other variants of `sqlite3_key()` function as well as those for the `sqlite3_rekey()` function and its variants are very similar and they are not shown here to avoid repetitions.

gaining momentum. As such, a `Plaintext extraction agent` for this library has been developed and included in our framework.

`Realm` provides both `Java` and `Kotlin` libraries but, because of space constraints, in this section we describe the `Java` library only.

The analysis of the documentation and of the source code of `Realm` [32] indicates that:

- `Realm` uses only an encryption key (and not additional parameters, as instead done by `SQLCipher`) to encrypt and decrypt data;
- to obtain the encryption key, it is sufficient to hook the constructor of the `RealmConfiguration` class, which is used to open an existing database;
- to obtain the plaintext data of an encrypted database, it is sufficient to hook the `close` method of the `Realm` class.

The resulting code for these hooks is reported in Fig. 11.

```
1  Java.perform(function x() {
2      const RealmConfiguration = Java.use("io.realm.RealmConfiguration")
3      const Realm = Java.use("io.realm.Realm")
4      const File = Java.use("java.io.File");
5      function dumpDbToPlainText(realmConfigInstance) {
6          const filePlaintext = File.$new(plaintextPath)
7          const instanceRealm =
         Realm.getInstance.overload('io.realm.RealmConfiguration').call(Realm,
         realmConfigInstance)
8          instanceRealm.sharedRealm.value.writeCopy(filePlaintext, null);
9      }
10     RealmConfiguration.$init.overload(
11         'java.io.File',
12         'java.lang.String',
13         '[B',
14         'long',
15         'io.realm.RealmMigration',
16         'boolean',
17         'io.realm.internal.OsRealmConfig$Durability',
18         'io.realm.internal.RealmProxyMediator',
19         'io.realm.rx.RxObservableFactory',
20         'io.realm.coroutines.FlowFactory',
21         'io.realm.Realm$Transaction',
22         'boolean',
23         'io.realm.CompactOnLaunchCallback',
24         'boolean',
25         'long',
26         'boolean',
27         'boolean')
28         .implementation = function (realmPath,
29         a1, key, a3, a4, a5, a6, a7, a8, a9,
30         a10, a11, a12, a13, a14, a15, a16) {
31         const toRet = this.$init(realmPath, a1, key,
32             a3, a4, a5, a6, a7, a8, a9,
33             a10, a11, a12, a13, a14, a15, a16);
34         Report_key_to_FRIDA_Client(key)
35         return toRet
36     };
37     Realm.close.overload().implementation = function () {
38         dumpDbToPlainText(this.sharedRealm.value.getConfiguration())
39     };
40 })
```

**Figure 11: `Realm`: hooking of the `RealmConfiguration` constructor and the `class` method.**

**4.2.1 Extracting plaintext contents** To extract the plaintext contents of the database, we hook the `close` method of the `Realm` class (lines 37–39 of Fig. 11) by overwriting it with the `dumpDbToPlainText` function (which is defined in lines 5–9). In this function, the object named `filePlainText` (line 6) stores the plaintext copy of the database, the input parameter `realmConfigInstance` is used to open the database (line 7) and, finally, the `writeCopy` function is invoked on the `sharedRealm` field contained in the Realm instance (line 8) to actually perform the plaintext copy of the database.

**4.2.2 Extracting the encryption key** The `JavaScript` code to hook the `RealmConfiguration` constructor is reported in lines 10–36 of Fig. 11. In particular, we first instantiate the name of the method to hook (lines 10 – 27), and the code executed by the hook (lines 28 – 36). The code of the hook first opens (and decrypts) the database (line 31), and then sends to the `FRIDA Client` the encryption key (by means of the generic function `Report_key_to_FRIDA_Client` (line 34)).

## 4.3 Decrypting the Android `Jetpack Security` library

The `Jetpack Security` library is part of the Android Jetpack suite of libraries and its main goal is to help developers follow security best practices related to securely reading and writing files, as well as key management through the Android Keystore system. Considering the wide adoption of Jetpack libraries in Android apps, we developed a `Plaintext extraction agent` for the `Jetpack Security` library and included it in our framework.

The `Jetpack Security` library provides an API both in Java and Kotlin programming languages but, since both APIs are quite similar in the functionality provided and in the interface, and due to space limits, in this paper we describe the Java API only.

`Jetpack Security` provides two classes to securely reading and writing data at rest, namely the `EncryptedFile` class (used to read and write encrypted files) and the `EncryptedSharedPreferences` class (used to encrypt keys and values in a preference file). Currently, our framework fully supports plaintext extraction from `EncryptedSharedPreferences` instances only, but we are working to support also the plaintext extraction from `EncryptedFile` instances (which we plan to present in a future work).

From the analysis of the documentation and of the source code of `Jetpack Security` [4], we found that to obtain the plaintext data of an encrypted preference file, it is sufficient to hook both the `create` method of the `EncryptedSharedPreferences` class (invoked to create an instance of this class) as well as the `commit` and `apply` methods of the `Editor` nested class (invoked to commit preferences changes from memory back to the preference file). The reason to hook all the above methods is to cover all possible experimental scenarios, including those where the preference file is not changed (in this case, neither the `commit` and `apply` methods will be invoked; therefore the plaintext extraction is performed in the `create` method), and also where the preference file is modified (in this case, hooking the `commit` and `apply` methods assures that the most updated plaintext is extracted).

The resulting code for these hooks is reported in Fig. 12. As we can note, each hook invokes the original (hooked) method and saves the returned value in an auxiliary variabile `ret`, then dumps the contents of the preference file in plaintext to the console through the `dumpSharedPrefs` function (defined at lines 1–10), and finally returns the value returned by the hooked function that we previously stored in the variable `ret`. For instance, in the hook for the `create` method, we first invoke the original `create` method and save the returned value (representing an instance of an encrypted `SharedPreferences`) in the variable `ret` (line 23), then we dump the preferences data in plaintext to the console by calling the dumpSharedPrefs function (line 24), and finally we return the instance of the encrypted SharedPreferences stored in the variable ret (line 25).

```
1  function dumpSharedPrefs(sp) {
2      var m = sp.getAll();
3      var HashMapNode = Java.use('java.util.HashMap$Node');
4      var iterator = m.entrySet().iterator();
5      console.log("Shared Preferences:");
6      while (iterator.hasNext()) {
7        var entry = Java.cast(iterator.next(), HashMapNode);
8        Report_params_to_FRIDA_Client(entry.getKey(),entry.getValue())
9      }
10 }
11
12 Java.perform(function() {
13 const EncryptedSharedPrefs =
         Java.use('androidx.security.crypto.EncryptedSharedPreferences');
14 const EncryptedSharedPrefsEditor =
         Java.use('androidx.security.crypto.EncryptedSharedPreferences$Editor');
15
16 EncryptedSharedPrefs.create.overload(
17  'android.content.Context',
18  'java.lang.String',
19  'androidx.security.crypto.MasterKey',
20  'androidx.security.crypto.EncryptedSharedPreferences$PrefKeyEncryptionScheme',
21 'androidx.security.crypto.EncryptedSharedPreferences$PrefValueEncryptionScheme')
22  .implementation = function(context, fileName, masterKey,
         prefKeyEncryptionScheme, prefValueEncryptionScheme) {
23      var ret = this.create(context, fileName, masterKey,
         prefKeyEncryptionScheme, prefValueEncryptionScheme);
24      dumpSharedPrefs(ret);
25      return ret;
26 }
27
28 EncryptedSharedPrefsEditor.apply.implementation = function() {
29      var ret = this.apply();
30      dumpSharedPrefs(this.mEncryptedSharedPreferences.value);
31      return ret;
32 }
33
34 EncryptedSharedPrefsEditor.commit.implementation = function() {
35      var ret = this.commit();
36      dumpSharedPrefs(this.mEncryptedSharedPreferences.value);
37      return ret;
38 }
39 });
```

**Figure 12: Jetpack Security: hooking of the EncryptedSharedPreferences class and of its Editor nested class.**

## 5 Experimental results

In order to validate the decryption framework we developed, we performed experiments in which the decryption parameters are extracted from real-world applications. [5] In these experiments, we consider a set of applications that use either SQLCipher or Realm to encrypt their databases, we install and initialize these apps on an Android device (to ease the experimentation we use virtual Android devices with the Android Emulator [10], but a rooted real Android device could have been used instead without changing anything in our experiments). For the experiments, we considered apps whose decryption procedure has been already published in the literature, so as to validate our framework against published results, and in particular Wickr Me (version 5.84.6), Signal (version 5.19.4), and Threema (version 4.8), whose decryption procedure has been published in [19]. Note that while for Signal and Wickr Me we use the same versions considered in [19], as these versions can still be downloaded from the APKMirror site [36], for Threema we use the version currently available on the Google Playstore as, to the best of our knowledge, no previous versions of this app are available on third-party sites like APKMirror.

[5]Note that we do not report here the plaintext of the data encrypted by these apps, since we had no way to compare them to existing published works. We stress however that, in this kind of validation experiments, our framework reported the correct plaintexts of the above data.

Since all these apps use SQLCipher, we consider also Element that, instead, uses Realm, although there are no published results for it against which our results can be compared.

Concerning the apps using SQLCipher, namely Wickr Me, Signal and Threema, with our framework we are able to extract the same encryption parameters presented in [19] (except for the encryption key, which is obviously different and thus not considered in the validation). This can be verified by comparing the values of the parameters extracted with our framework and reported in Table 2 (where, for each encryption parameter, we report its values extracted from the encrypted database generated by the above apps) with those presented in [19]. For completeness, we also report the encryption keys (truncated to avoid cluttering) extracted by our framework, which however have been placed outside the main table, as these values are, of course, different from those reported in [19], as they have been created on a device different from those used in the above paper.

We also performed some experiments to validate the results obtained with the Realm Plaintext extraction agent. Given that, to the best of our knowledge, there are no published results concerning the decryption of Realm databases, we performed experiments using Element [11], a secure messaging app which uses the Realm library to manage encrypted databases. In particular, we tested our agent with Element version 1.4.3. In our experiments we installed Element on a virtual device and we populated its databases with data by using the app to exchange messages. Then we instrumented it with the Realm Plaintext extraction agent and, by using our framework, we found out that it uses several encrypted databases, which in all cases but two use different encryption keys. The list of the encryption keys extracted are reported in Table 3. In order to validate our agent, we use the Realm Studio [33], a developer tool to easily manage Realm databases. In particular, we pass to Realm Studio the encrypted databases and the encryption keys extracted by our agent and we successfully obtain the plaintext database contents.

## 6 Conclusions

In this paper, we have presented a framework for the decryption of data encrypted by apps, which is based on the use of dynamic instrumentation of app's binary code by means of hooking. Our framework has been conceived to be used only with test devices used for forensic study purposes, and not with devices that need to be forensically analyzed.

By executing suitable hooks when the app under study is executed on a test device, our framework can export the plaintext of data after they have been decrypted by the app, as well as the corresponding encryption key and parameters (when possible), thus enabling the experimenter to access and analyze them.

Hooking works at the function level, meaning that once the hook for a given decryption function has been developed for the first time, it can be used with any app using the same function. Therefore, by writing hooks for popular encryption libraries, it is

| *Parameter* | `Wickr Me` (database: `wickr_db`) | `Signal` (database: `signal.db`) | `Threema` (database: `threema4.db`) |
|---|---|---|---|
| `kdf_iter` | 256000 | 1 | 1 |
| `cipher_page_size` | 4096 | 4096 | 4096 |
| `cipher_use_hmac` | 1 | 1 | 1 |
| `cipher_plaintext_header_size` | 0 | 0 | 0 |
| `cipher_hmac_algorithm` | HMAC_SHA512 | HMAC_SHA1 | HMAC_SHA512 |
| `cipher_kdf_algorithm` | PBKDF2_HMAC_SHA512 | PBKDF2_HMAC_SHA1 | PBKDF2_HMAC_SHA512 |
| `encryption key` | `0x3030316432353861653...` | `0x6361376461316539303...` | `0x7822393131656362337...` |

**Table 2: Main encryption parameters extracted with our framework from the encrypted database generated by `Wickr Me`, `Signal` and `Threema`. Their values (except for the encryption keys) are the same one reported in [19].**

| | |
|---|---|
| `matrix-sdk-auth.realm` | 0x7878C1FE4364FB8867BE645751917826CDE26C7A1CBDD... |
| `disk_store.realm` | 0xF4C024F59FA71E203B8EC24DE83CF80A05CAA15840162... |
| `crypto_store.realm` | 0xA35E9873554F56286527FD9926E957829DBC8F05BE1EA... |
| `matrix-sdk-identity.realm` | 0xF4C024F59FA71E203B8EC24DE83CF80A05CAA15840162... |
| `matrix-sdk-content-scanning.realm` | 0xF4C024F59FA71E203B8EC24DE83CF80A05CAA15840162... |
| `matrix-sdk-global.realm` | 0xDDC9411EEBC27CDA10AA8398134D899A5C29D1CC538B8... |

**Table 3: List of encryption keys extracted by the `Realm Plaintext extraction agent`. Keys are truncated due to space constraints.**

possible to support the decryption of data for all apps that use these libraries. Our framework currently supports two prominent and popular encryption libraries, namely `SQLCipher` [37] and `Jetpack Security` [3], and the `Realm` [34] DBMS. We have validated it by comparing our results with those reported in literature for several real-word apps that use encryption.

As future work, we plan to expand the set of encryption libraries supported by the framework, as well as to support also iOS.

## References

[1] Android. 2022. The Room persistence library. https://developer.android.com/jetpack/androidx/releases/room Accessed on Oct 7, 2022.
[2] Android. 2023. The javax.crypto encryption class. https://developer.android.com/reference/javax/crypto/package-summary Accessed on Feb 2, 2023.
[3] Android. 2023. The Android Jetpack Security Library. https://developer.android.com/jetpack/androidx/releases/security Accessed on Feb 2, 2023.
[4] Android. 2023. The androidx.security.crypto API reference. https://developer.android.com/reference/androidx/security/crypto/package-summary Accessed on Feb 2, 2023.
[5] C. Anglano. 2014. Forensic Analysis of WhatsApp Messenger on Android Smartphones. *Digital Investigation* 11, 3 (Sept. 2014), 201–213.
[6] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. 2016. Forensic Analysis of the ChatSecure Instant Messaging Application on Android Smartphones. *Digital Investigation* 19 (Dec. 2016), 44–59.
[7] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. 2017. Forensic analysis of Telegram Messenger on Android smartphones. *Digital Investigation* 23 (2017), 31–49.
[8] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. 2020. The Android Forensics Automator (AnForA): A tool for the Automated Forensic Analysis of Android Applications. *Computers & Security* 88 (2020).
[9] SQLite Consortium. 2023. SQLite Home Page. https://www.sqlite.org Accessed on Mar 13, 2023.
[10] Google Developers. 2022. Run apps on the Android Emulator. https://developer.android.com/studio/run/emulator Accessed on Oct 8, 2022.
[11] Element. 2022. Element | Secure Collaboration and Messaging . https://element.io/ Accessed on Oct 7, 2022.
[12] Element. 2022. Element Android build.gradle. https://github.com/vector-im/element-android/blob/develop/matrix-sdk-android/build.gradle Accessed on Oct 7, 2022.
[13] Andrew Case et al. 2019. HookTracer: A System for Automated and Accessible API Hooks Analysis. *Digital Investigation* 29 (2019), S104–S112.
[14] D. Votipka et al. 2020. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *29th USENIX Security Symposium*.
[15] Giyoon Kim et al. 2020. A study on the decryption methods of telegram X and BBM-Enterprise databases in mobile and PC. *Forensic Science International: Digital Investigation* 35 (2020).
[16] Giyoon Kim et al. 2021. Forensic analysis of instant messaging apps: Decrypting Wickr and private text messaging data. *Forensic Science International: Digital Investigation* 37 (2021), 301138.
[17] H. Zhang et al. 2018. Digital Forensic Analysis of Instant Messaging Applications on Android Smartphones. In *2018 International Conference on Computing, Networking and Communications (ICNC)*. 647–651.
[18] Jusop Choi et al. 2019. Digital forensic analysis of encrypted database files in instant messaging applications on Windows operating systems: Case study with KakaoTalk, NateOn and QQ messenger. *Digital Investigation* 28 (2019).
[19] Jihun Son et al. 2022. Forensic analysis of instant messengers: Decrypt Signal, Wickr, and Threema. *Forensic Science International: Digital Investigation* 40 (2022), 301347.
[20] L. Zhang et al. 2016. The Forensic Analysis of WeChat Message. In *2016 Sixth International Conference on Instrumentation Measurement, Computer, Communication and Control (IMCCC)*. 500–503.
[21] Myungseo Park et al. 2019. Decrypting password-based encrypted backup data for Huawei smartphones. *Digital Investigation* 28 (2019).
[22] Myungseo Park et al. 2020. A methodology for the decryption of encrypted smartphone backup data on android platform: A case study on the latest samsung smartphone backup system. *Forensic Science International: Digital Investigation* 35 (2020).
[23] Soojin Kang et al. 2021. Methods for decrypting the data encrypted by the latest Samsung smartphone backup programs in Windows and macOS. *Forensic Science International: Digital Investigation* 39 (2021).
[24] Sumin Shin et al. 2022. Forensic analysis of note and journal applications. *Forensic Science International: Digital Investigation* 40 (2022).
[25] Tobias Groß et al. 2019. Analyzing Android's File-Based Encryption: Information Leakage through Unencrypted Metadata. In *Proceedings of the 14th International Conference on Availability, Reliability and Security* (Canterbury, CA, United Kingdom) *(ARES '19)*. Association for Computing Machinery, New York, NY, USA.
[26] Yu-an Tan et al. 2020. An Android Inline Hooking Framework for the Securing Transmitted Data. *Sensors* 20, 15 (2020).
[27] Yu Zhang et al. 2020. Android Encryption Database Forensic Analysis Based on Static Analysis. In *Proceedings of the 4th International Conference on Computer Science and Application Engineering* (Sanya, China) *(CSAE 2020)*. Association for Computing Machinery, New York, NY, USA.
[28] Frida. 2022. Frida: a world-class dynamic instrumentation framework. https://frida.re/ Accessed on Oct. 5th, 2022.
[29] Google Developers. 2022. Android NDK. https://developer.android.com/ndk Accessed on Oct 6, 2022.
[30] iBotPeaches. 2022. Apktool - A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/ Accessed on Oct 7, 2022.
[31] T. Mehrotra and B. M. Mehtre. 2013. Forensic analysis of Wickr application on android devices. In *2013 IEEE International Conference on Computational Intelligence and Computing Research*. 1–6.
[32] MongoDB. 2022. Encrypt a Realm - Java SDK. https://www.mongodb.com/docs/realm/sdk/java/advanced-guides/encryption/ Accessed on Oct 10, 2022.
[33] MongoDB. 2022. Realm Studio. https://www.mongodb.com/docs/realm-legacy/products/realm-studio.html Accessed on Oct 13, 2022.
[34] MongoDB. 2022. Realm.io. https://www.realm.io Accessed on Oct 10, 2022.
[35] P. Cobley and G. Geneste. 2022. Realm. In *Mobile Forensics - The File Format Handbook*, C. Hummert and D. Pawlaszczyk (Ed.). Springer, Chapter Chapter 8.
[36] Wickr. 2022. APKMirror - Free APK Downloads - Free and safe Android APK downloads. https://www.apkmirror.com/ Accessed on Oct 7, 2022.
[37] Zetetic. 2022. SQLCipher. https://www.zetetic.net/sqlcipher/ Accessed on Oct 9, 2022.
[38] Zetetic. 2022. SQLCipher for Android. https://github.com/sqlcipher/android-database-sqlcipher Accessed on Oct 7, 2022.