One Key to Rule Them All: Recovering the Master Key from RAM to break Android's File-Based Encryption

By:
Tobias Groß, Marcel Busch and Tilo Müller

From the proceedings of
The Digital Forensic Research Conference
DFRWS EU 2021
March 29 - April 1, 2021

Forensic Science International: Digital Investigation 36 (2021) 301113

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

Full Paper

One key to rule them all: Recovering the master key from RAM to break Android's file-based encryption

Tobias Groß*, Marcel Busch, Tilo Müller

Friedrich-Alexander University (FAU), Erlangen, Nürnberg, Germany

ARTICLE INFO

Article history:
Available online 23 March 2021

Keywords:
Android
EXT4
File-based encryption (FBE)
Disk forensics
Memory forensics
Cold boot attacks

ABSTRACT

As known for a decade, cold boot attacks can break software-based disk encryption when an attacker has physical access to a powered-on device, including Android smartphones. Raw memory images can be obtained by resetting a device and rebooting it with a malicious boot loader, or—on systems where this is not possible due to secure boot or restrictive BIOS settings—by a physical transplantation of RAM modules into a system under the control of the attacker. Based on the memory images of a device, different key recovery algorithms have been proposed in the past to break *Full Disk Encryption* (*FDE*), including BitLocker, dm-crypt, and also Android's FDE. With Google's switch from FDE to *File-based Encryption (FBE)* as the standard encryption method for recent Android devices, however, existing tools have been rendered ineffective. To close this gap, and to re-enable the forensic analysis of encrypted Android disks, given a raw memory image, we present a new key recovery method tailored for FBE. Furthermore, we extend *The Sleuth Kit (TSK)* to automatically decrypt file names and file contents when working on FBE-enabled EXT4 images, as well as the *Plaso* framework to extract events from encrypted EXT4 partitions. Last but not least, we argue that the recovery of master keys from FBE partitions was particularly easy due to a flaw in the key derivation method by Google.

1. Introduction

When a turned-on but locked smartphone is lost, stolen, or seized, it faces attacks exploiting the capabilities of having physical access to it. In 2008, Halderman et al. (2008) presented a method to break FDE, including BitLocker, TrueCrypt and FileVault, called the *cold boot attack*. This attack is based on the *remanence effect* of DRAM, which says that memory modules preserve their contents for a short time after power is cut. This time can be extended, from less than a second up to several minutes, if the RAM modules get cooled, either by cooling sprays or by putting the device into a freezer (Müller and Spreitzenbarth, 2013). After cooling, two different kinds of cold boot attacks can be enforced: Either the target machine is reset and booted with a forensic boot loader to recover encryption keys from RAM. In this case, power is cut only briefly, and the rate of correct recovered bits is high. Or, if the target machine has boot restrictions such as secure boot or BIOS settings, RAM modules must quickly be transplanted into a recovery machine under the control of the attacker. In the latter case, power is cut for several seconds, and the rate of successfully recovered bits depends on the temperature of the memory modules, as well as other physical properties of DRAM.

Following the work by Halderman et al., more academic studies have been published that substantiate the practicability of cold boot attacks against common desktop PCs (Gruhn and Müller, 2013) as well as Android-driven smartphones (Müller and Spreitzenbarth, 2013). With DDR3 *memory scrambling* was introduced, rendering cold boot attacks difficult because memory scrambling uses an LFSR-based algorithm to "encrypt" RAM. However, it was never meant for encryption but for the equal distribution of ones and zeroes, just to increase efficiency. As a consequence, even though scrambling is initialized differently on each boot, subsequent work could show how to break memory scrambling to access plain RAM (Bauer et al., 2016).

On the defending side, to protect FDE keys and other crypto keys, specialized key storages have been proposed to hold the keys in CPU registers only, not in RAM (Müller et al., 2011; Garmany and Müller, 2013). But those systems are pure academic concepts that are not used in productive environments, as far as we know. And second, such systems typically protect only one key, like the FDE key, while other contents remain unencrypted in RAM. Hence, cold boot attacks not focusing on cryptographic keys but other memory contents, such as images and messages on Android devices, remain a threat (Hilgers et al., 2014). Consequently, more sophisticated countermeasures

* Corresponding author.
E-mail addresses: tobias.gross@cs.fau.de (T. Groß), marcel.busch@cs.fau.de (M. Busch), tilo.mueller@cs.fau.de (T. Müller).

https://doi.org/10.1016/j.fsidi.2021.301113

encrypted large parts of the RAM, either in software (Götzfried et al., 2016a, 2016b) or in hardware (Würstlein et al., 2016), but also those systems stay academic concepts that are not used on real-world devices for a simple reason: Memory encryption slows down the overall system performance by an order of magnitude.

To sum up, also ten years after, the security hole opened by cold boot attacks could not be closed in general yet. Indeed, on ARM-driven Android devices with hardware-backed key storages and secure boot enabled (*TrustZone*), the exploitation of the remanence effect is extremely difficult today. Due to restrictive platform settings enforced by Android vendors, not allowing to start custom boot loaders on locked devices, the transplantation of RAM modules often remains the only way to obtain memory dumps. This method, however, is not easily possible on embedded devices like smartphones, because RAM modules are soldered onto the board and not unpluggable.

Those, for adversaries on the lower end of available resources, like individual security researchers, it becomes unaffordable today to obtain raw memory dumps from up-to-date, fully patched Android smartphones. Contrary to that, adversaries on the upper end of available resources, like state-level actors, have multiple options: BootROM exploits for zero-day vulnerabilities (Checkra1n Jailbreak: Anal, 2020; exynos-usbdl: unsigned co, 2020), as well as "secret" boot loader stages from the vendors (Redini et al., 2017). Costs do not matter for state-level adversaries, vendors can often be forced to co-operate, and chip-off attacks are an established way to carry out investigations (Mikhaylov, 2016).

1.1. Contributions

The above-mentioned restrictions issuing from platform security hold true for all FBE-enabled Android devices we had access to, most notably the Nexus and Pixel series by Google. Consequently, we cannot present the full chain of a cold boot attack against an Android device with FBE when it is fully patched and has its security features enabled. Nevertheless, we argue that our work is an important *building block* in the area of key recovery and memory forensics.

Also, our work is of interest from a disk forensics point of view. Not only rogue attackers have an interest in breaking disk encryption to steal private data, but also law enforcement has an eligible interest in decrypting smartphones for forensic analyses. Today, smartphones are an integral part of our lives, storing most of our personal and business data, such that authorities world-wide have an increasing interest in accessing the data of suspects to prove or disprove facts of a crime.

In detail, our contributions are as follows:

- Given a raw memory image, we developed a method that recovers the master key from file keys on Android EXT4 FBE partitions.
- We extended *The Sleuth Kit* (TSK) (Carrier, 2020) to output FBE-related file attributes and to decrypt all file names and file contents automatically.
- We extended the Plaso framework (Welcome to the Plaso docu, 2020) to be able to extract events from an FBE-encrypted partition when providing only the master key.
- We evaluated our method on 13 Android smartphones, realease between 2015 and 2020, showing that 7 out of them use a key derivation function that is vulnerable to our method.

Finally, we argue that the recovery of master keys from FBE was particularly easy due to a flaw in the key derivation function (KDF) used. Google independently fixed this issue in newer kernel versions. However, the KDF is not changed when updating a smartphone over-the-air, as it requires re-encryption of the whole device, such that older phones remain vulnerable.

1.2. Related work

In 2008, Halderman et al. (2008) presented the `aeskeyfind` tool to recover FDE keys from RAM. This tool is based on an algorithm that identifies the AES key schedule structure in memory. The tool alone, however, cannot be used to recover the master key to FBE partitions.

In 2017, Loftus et al. (2017) gave an overview of Android's FBE and investigated whether known attacks against FDE are still applicable. As a result, they state that some attacks appear still feasible on Android 7.0 and later. In this paper, we particularly show how the key recovery method and forensic toolchains must be adapted at times of FBE.

In 2019, Groβ et al. (2019) investigated whether sensitive data about the usage behavior of smartphone users can leak when using Android's FBE feature. Metadata remained unencrypted, and indeed, the authors came to the conclusion that the metadata is sufficient to extract a list of installed apps and to recover traces of usage behavior, such as the time when WhatsApp messages have been sent or received. The attack shown by Groβ et al. (2019), however, was limited to basic events that could be extracted from an unencrypted FBE partition. In this paper, we show that FBE can be broken entirely after a RAM image was obtained.

In 2016, Unterluggauer and Mangard (2016) investigated different disk encryption schemes to gain the encryption key with differential power analysis and differential fault analysis, rather than from RAM. They checked the FDE implementations of Android, Mac OS X, and Linux (dm-crypt), including the FBE feature of EXT4. For FBE they mentioned that it is possible to recover the master key from one file-specific data encryption key and the file-specific nonce – a fact we will use later on, as well.

To sum up more related work: Skillen et al. (2013), Wang et al. (2012) and Teufl et al. (2014) evaluated the security of Android's FDE and proposed novel methods for hardening FDE. (Müller and Spreitzenbarth, 2013) implemented a forensic boot loader called FROST that extracts the FDE key from a cold booted Android device. To overcome attacks like FROST, Götzfried and Müller (2014) implemented a method that hardens Android's FDE against memory forensics attacks by storing FDE keys in CPU registers.

2. Background

In this section, we give necessary background information on FBE (Section 2.1), as well as the forensic tools *TSK* (Section 2.2) and Plaso (Section 2.3).

2.1. Android file-based encryption

With Android 7.0, Google introduced FBE as a possible replacement for FDE. With Android 10.0, FBE gets the mandatory encryption scheme for new devices. FBE encrypts user files rather than a whole partition, such that individual files can remain unencrypted. As a consequence, basic features of a smartphone (e.g., alarm clocks, receiving calls) can be running without unlocking the phone first, and the files of different user accounts get encrypted differently. As no entire partitions get encrypted, FBE is implemented as an EXT4 feature that encrypts filenames and file contents.

Only with additional support on recent Android phones, also metadata gets encrypted. Therefore, with Android 9.0, Google introduced support for metadata encryption (Metadata Encryption, 2020). With metadata encryption enabled, FDE is used on top of FBE to encrypt the whole partition. Hence, for decrypting file data, one has first to decrypt the FDE partition and afterward, the file contents encrypted by FBE. Typically that metadata encryption FDE is performed with a device bound key to still be able to start basic services at boot time without unlocking the device first.

Note that plenty of related work has already been published on breaking FDE with cold boot attacks. As a consequence, the combination of FDE and FBE makes our attack still valuable as a part of the decrypting chain on recent devices. First, FDE would have to be broken, e.g., by approaches like by Halderman et al. (2008), and subsequently, FBE would have to be broken by our approach.

Technically, with FBE, each folder encrypts the including filenames with a separate key, and analogously, every file content is encrypted with a different key. According to Unterluggauer and Mangard (2016), we name the data encryption key to a file or folder DEK_f. The DEK_f keys are derived from a master key *MK*. On a typical Android system, there are at least two different types of master keys. One that is bound to the device (notated as MK_d) and is used to encrypt files that should be accessible after booting the device without unlocking it with a pin code. Typically the device binding of such a master key is implemented with the help of the *ARM TrustZone*, which keeps the key save from root- and kernel-level attackers. Other master keys are derived from a user's credentials (e.g., MK_c). This derivation is done in the *TrustZone* as well, in order to ensure that attackers cannot brute-force weak pin codes easily. The MK_c key is used to encrypt files, which should be more confidential, like chat histories, for example.

Every encrypted file and folder in EXT4 specifies with a key descriptor which master key should be used to derive the DEK_f. A key derivation function (*KDF*) outputs the DEK_f by using the inputs *MK* and a file specific nonce.

Fig. 1 shows the usage of the different keys when accessing the content of a file. In step ①, access to a file is requested. With the help of the key descriptor of the file, the suitable master key *MK* gets selected ②. The file nonce and the master key is then used to derive the DEK_f with the help of the key derivation function ③. With the derived DEK_f, the file content can be decrypted ④ with the decryption function (DF).

2.2. The Sleuth Kit

TSK is an open-source library and tool collection for the forensic analysis of file systems and partitions (Carrier, 2020). Its functionality is divided into multiple modules, each focusing on a storage layer (e.g., image, partition, file system) to make an extension as easy as possible.

TSK allows for performing a physical acquisition where a bit-by-bit copy of the smartphone's storage gets analyzed. This technique potentially allows for the recovery of deleted and hidden data. Contrary to the physical acquisition is the logical acquisition, where the operating system (OS) of the smartphone is used for data acquisition (Sammons, 2012). Drawbacks of this method include that no deleted data can be recovered and that data can be altered by accident or inevitable because the OS and running applications constantly access files in the background.

For every task which is related to the file system or the partition analysis, TSK provides a single purpose command-line tool. For example, *istat* is used to show the metadata of a file, and *icat* is used to extract the content data of a file. There is also a tool to list the files contained in a folder called *fls*. There are many other tools that can handle, for example, journal data, block-level access, or access to data on the partition level.

Naturally, the functionality of the above-mentioned tools in TSK is affected when FBE is enabled in EXT4. With FDE, it is possible to separate the decryption functionality from the file system analysis and, therefore, TSK would have no need for decryption. But with FBE, encryption is coupled tight to the filesystem, and therefore, it is required to extend TSK with decryption functionality to be able to efficiently analyze and extract data from EXT4 FBE filesystems.

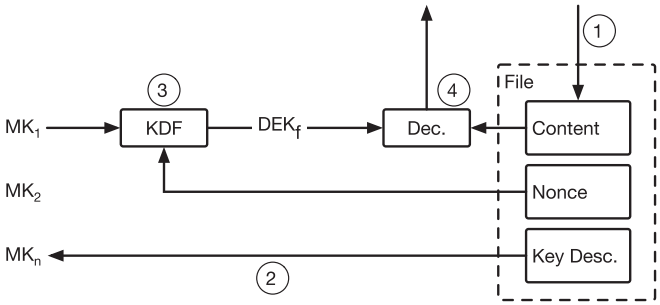

Fig. 1. File-based decryption process when accessing the content of a file.

2.3. Plaso

The *Plaso Project* is the successor of the *log2timeline* tool. The tools of this project allow us to create a super timeline, where all timestamped data of a system is collected in one timeline (Welcome to the Plaso docu, 2020). Such a super timeline is a good base for event reconstruction in forensic applications.

Plaso iterates over different storage layers like partitions, it's file-systems, and application files to extract timestamped data. The file system level access functionality is encapsulated in the digital forensic virtual file system (dfVFS) project (Digital Forensics Virtual, 2020). Typically, timestamped data can be found in log files, instant messengers, and browser databases, for example, but also in the metadata of files. For many file types, Plaso provides a parser and special analyzer for different application data. For example, the history of the Chrome browser is stored as an SQLite database. Plaso provides an SQLite format parser module and a special analyzer that knows how to interpret the entries in different tables of the history database.

As we already argued for TSK, Plaso also needs to be extended to be able to handle FBE encrypted images. With FDE, there was no need for Plaso to handle encrypted images because partitions could be decrypted beforehand. With FBE, however, the encryption is tightly coupled to the files, such that we need to extend Plaso and dfVFS to be able to handle FBE encrypted files.

3. File-based encryption attack

As an attacker model, we choose an attacker who has *physical access* to an Android device with FBE. This is not a limitation as the task of disk encryption is to prevent attackers who have physical access from gaining data that is persistently stored on a device. Additionally, we assume the device is powered on, but it might be locked, which is a typical state for many smartphones. Last but not least, and this is a limitation, we need full access to the RAM image of a device (and, of course, access to its encrypted disk). Only when given a full memory image we can exploit a weakness in the key derivation function of FBE to obtain the master key that can eventually be used to decrypt the stored data.

3.1. Prerequisites

For our attack, we need (1) a memory image and (2) naturally a copy of the user data partition from the attacked device. There are different possibilities to obtain these images, which we elaborate on in the following. But note that in general, no Android Debug Bridge (ADB) or other pre-installed developer access to the phone is required. In general, a RAM image must "somehow" be obtained for our attack. This can, as an alternative to the above mentioned cold boot attacks—which either require the bootloader to be unlocked or the RAM modules to be unpluggable—by exploits and other tweaks that allow the execution of code in an early boot phase.

```
1  static int derive_key_aes(u8 deriving_key[FS_AES_128_ECB_KEY_SIZE],
2                            const struct fscrypt_key *source_key,
3                            u8 derived_raw_key[FS_MAX_KEY_SIZE])
4  {
5          /* ... */
6          struct crypto_skcipher *tfm = crypto_alloc_skcipher("ecb(aes)", 0, 0);
7          /* ... */
8          res = crypto_skcipher_setkey(tfm, deriving_key,
9                                       FS_AES_128_ECB_KEY_SIZE);
10         /* ... */
11         sg_init_one(&src_sg, source_key->raw, source_key->size);
12         sg_init_one(&dst_sg, derived_raw_key, source_key->size);
13         skcipher_request_set_crypt(req, &src_sg, &dst_sg, source_key->size,
14                                    NULL);
15         res = crypto_wait_req(crypto_skcipher_encrypt(req), &wait);
16         /* ... */
17         return res;
18 }
```

Listing 1. Implementation of the key derivation function (KDF) in the Android kernel source.

3.1.1. Memory images

We know of at least two ways that, in principle, allow for obtaining memory contents from powered-on devices by the execution of code in an early boot phase.

Bootloader Stages Modern mobile devices pass through multiple bootloader stages until, eventually, the operating system, hosting the user-facing applications, gets started. The first bootloader stage is immutably stored on the device during the manufacturing process and referred to as *BootROM*. All succeeding stages are mutable but verified using cryptographic signatures, resulting in a trusted boot chain. A typical design of this trusted boot chain is to let bootloader stages drop their privileges according to their task. For instance, the BootROM begins on the highest privilege level, and privileges are dropped before control is transferred to the operating system (Dent, 2020).Manufacturers usually equip their chips with alternative boot modes that can be entered after a warm reset of the device. The boot mode entered also determines the privilege level, as explained before. While the *recovery* mode and the *fastboot* mode are commonly known, and could already provide direct access to RAM, there exist further undocumented and powerful early-stage boot modes. On HiSilicon chips, used on all Huawei devices, an early bootloader stage offers a serial console to transfer and boot signed code (hisi-idt, 2020). On Qualcomm chips, used on all Nexus and Pixel devices, a boot mode referred to as EmergencyDownload(EDL) mode allows for deploying so-called EDL programmers that directly grant access to RAM (Hay and Hadad, 2018a). These programmers need to be signed by the manufacturer as well, but many of them are leaked on the internet.

Bootloader Exploits While for the previously mentioned undocumented bootloader stages, cryptographically signed code is needed, an exploit in any of the early bootloader stages grants the same access. Provided a warm reboot of a device into a vulnerable bootloader stage allows for obtaining residual memory areas after successful exploitation. This scenario is not only realistic but even unpatchable if the very first bootloader stage, the BootROM, is affected, as recently demonstrated on Apple products with the checkm8 exploit (Checkra1n Jailbreak: Anal, 2020). A similarly severe flaw affected Samsung's flagship series *Galaxy*, where at least the Samsung Galaxy S7 model contains a flawed BootROM (exynos-usbdl: unsigned co, 2020). On this note, the research by Redini et al. (2017) underlines that the bootloader code has to be considered regarding the attack surface of mobile devices. In their research, they found multiple flaws in bootloader stages used on Huawei devices.

3.1.2. User data partition

To get an image of the persistent user data partition, an attacker has at least two options. The first option is to use a bootloader stage offering features to dump partitions from persistent memory, as described above. This is the case for many EDL programmers (Hay and Hadad, 2018b), as discussed previously.

The other option is to chip-off the data storage chip. The Nexus 6P uses eMMC flash memory, for example. The content of this type of memory can be dumped by using the method shown by Etemadieh et al. (2017). Another option for persistent data storage is NAND chips with similar methods for data extraction (Mikhaylov, 2016).

3.2. Master key derivation

Our attack is based on the fact that the KDF used in Android EXT4 FBE takes non-secret data as an encryption key. In the FBE implementation we investigated, AES256-ECB is used as KDF. Listing 1 shows the relevant code which derives a DEK_f from an MK. It is part of the Android kernel, which is open source. The code is accessible in the Google Source repository.[1] The shown code excerpts are from the file `fs/crypto/keyinfo.c` at the commit tag ASB-2018-12-05_4.14-p-release.

The function `derive_key_aes` gets indirectly called from the function `fscrypt_get_encryption_info` with some functions in between. In this call, the file nonce is taken as `deriving_key` parameter, and the master key is part of the parameter `source_key`. The function `fscrypt_get_encryption_info` is used from the EXT4 filesystems to derive the DEK_f among other things. Analogously, an identical function is used for the F2FS filesystem to derive DEK_f keys.

Listing 1 shows that the `deriving_key` (the nonce) is taken as deriving key and `source_key` (the master key) as source key. The last parameter only defines the destination for the derived key. If we investigate the `derive_key_aes` function, we see that AES-ECB is used as the cipher (line 6). Furthermore, we see that the `deriving_key` parameter is used as the encryption key (line 8).

We notate the AES ECB encryption of the master key with the file nonce to get the file-specific data encryption key as $DEK_f = AES^{ECB}_{nonce_f}(MK)$. With AES, decryption is the same function as

[1] https://android.googlesource.com/kernel/common.

```
1  static int ext4_derive_key(const struct ext4_encryption_context *ctx,
2                             const char master_key[EXT4_MAX_KEY_SIZE],
3                             char derived_key[EXT4_MAX_KEY_SIZE])
4  {
5          /* ... */
6          if (ctx->filenames_encryption_mode == EXT4_ENCRYPTION_MODE_AES_256_HEH)
7                  return ext4_derive_key_v2(ctx->nonce, master_key, derived_key);
8          else
9                  return ext4_derive_key_v1(ctx->nonce, master_key, derived_key);
10 }
```

Listing 2. Alternative implementation of the key derivation function (KDF) in the Android sources.

encryption. We can compute the master key as $MK = AES^{ECB}_{nonce_f}(DEK_F)$, which means we decrypt the file-specific decryption key with the inherent nonce.

In later versions of the Android kernel, a second KDF was implemented for EXT4 FBE. In this version, the KDF is chosen based on the filename encryption mode. If the mode `EXT4_ENCRYPTION_MODE_AES_256_HEH` is used, the new KDF is used. This decision is made in the function `ext4_derive_key` shown in Listing 2. This code is part of the file `fs/ext4/crypto_key.c` in the repository https://android.googlesource.com/kernel/msm commit 9db9532a.[2] This KDF uses the master key as the key for KDF, and our attack method is not directly applicable anymore. Currently, as can be seen in Section 5, the new KDF is only used by one of our evaluated phones.

Our attack to compute the master keys of an FBE filesystem uses file specific AES keys DEK_f that we find in the memory dump and file specific nonces which we find in the dump of the persistent user data. To extract DEK_F keys from memory we use the existing `aeskeyfind`[3] tool which locates AES keys. We decided to use this approach instead of parsing kernel structures because it is more resistant against partial overwriting or fading of memory, which can happen when cold booting a device.

To extract all file nonces from the user data partition, we use our extended TSK, meaning we parse the EXT4 filesystem. With all found DEK_f keys notated as set $FK = \{DEK_1, DEK_2, \ldots, DEK_n\}$ and all nonces present on the filesystem $N = \{nonce_1, nonce_2, \ldots, nonce_n\}$, we can calculate the set of all potential master keys $M = \{MK_1, MK_2, \ldots, MK_3\}$ by applying the AES ECB decryption:

$$AES^{ECB}_n(fk) : FK, N \rightarrow M$$

The valid master keys are a subset of the set M. The properties of the AES cipher ensures that decrypting data with the wrong key output's random data. Applied to our solution, this means that if we find two different pairs of $FK \times N$ that output the same master key MK, we have found a valid used master key.

This solution requires only two DEK_f keys derived from the same master key to be present in the memory dump to compute the corresponding MK. The master key allows us to derive all DEK_f and to decrypt all file names and file content of the EXT4 file-based encrypted filesystem.

Special attention is needed in the case of FBE because the master keys and the file-specific keys are of size 512-bit. This is only a pseudo key size used by some AES modes (f.e. AES XTS), which consists of 2 concatenated 256-bit keys (Ligh, 2014). AES can operate with key sizes of 256-bit max. All 512-bit keys present in FBE are two concatenated 256-bit keys. *aeskeyfind* and our computation outputs only half of these 512-bit keys, but we can concatenate the master key halves to get full 512-bit master keys and verify the concatenations by decrypting data on the filesystem.

4. Implementation

We implemented a new tool, called *fbekeyrecover*, that implements the method described above in Section 3.2. Additionally, TSK is extended to output FBE related metadata and to be able to decrypt file names and content when provided with the master keys. Also, we extended Plaso to be able to use the extended TSK, such that event reconstruction can be performed on Android devices with FBE. Fig. 2 shows all tools extended or implemented by us and how they interact together to extract events from an Android phone with FBE.

All code we implemented is made open-source and publicly available at https://www.cs1.tf.fau.de/research/system-security-group/one-key-to-rule/.

4.1. The Sleuth Kit extension

Our extension of TSK can be divided into two parts. First is the extension of the command line tool *istat* for EXT4 filesystems. This tool prints all information and metadata of a given metadata address (also known as inode). With our extension, the tool additionally outputs encryption-related metadata.

The second extension implements the possibility for TSK to decrypt file names and content of encrypted EXT4 partitions when the master keys are provided.

The *istat* tool calls internally the filesystem specific *istat* function which is implemented for every supported fileystem. For encryption metadata output we extended and refactored the `ext2fs_istat` function. In EXT4, the new encryption specific metadata is stored as an extended attribute (XA), which is an extension feature of EXT. There are two different types of XA. The first one is an internal XA, and it is stored inside of an inode metadata structure. The second version of XA is stored in a separate block, and the block is referenced inside the inode structure. These XAs are called external. EXT4 allows the encryption metadata to be stored as internal or external XAs.

We excluded the parsing of XAs, which is already present in the istat function into a new function `ext2fs_print_xattr` and extended it, to be able to call it twice for the internal and external extended attribute data. We identify encryption XA with its specific attribute type (idx) of `EXT2_EA_IDX_ENCRYPTION` (=9). All EXT4 constants and structures definitions from our extension are taken from the Android kernel source. The encryption XA is defined in the `ext2fs_encryption_entry` structure. It contains the values version, content encryption mode, name encryption mode, key descriptor (8 bytes), and nonce (16 bytes). Content and name encryption mode define which cipher was used to encrypt. The key

[2] 9db9532ae32d13333d5f394e5775fb99fafaad45.
[3] https://github.com/makomk/aeskeyfind.

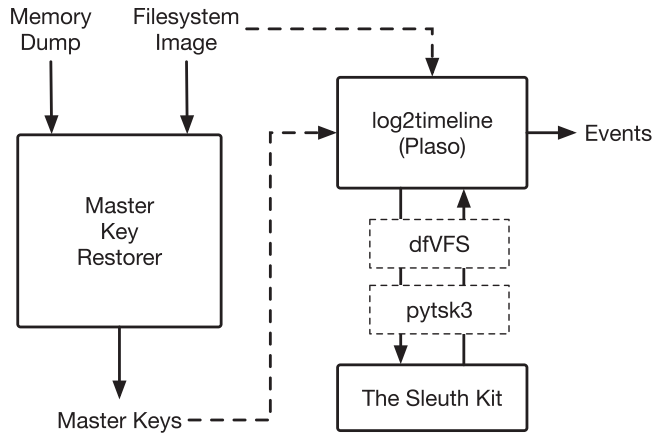

Fig. 2. Overview of the modules and tools we extended or implemented and how they interact with each other.

descriptor defines which master key must be used to derive the DEK_f key. The nonce is used in the derivation process. The Android kernel defines 6 (with the invalid option 7) different encryption modes: invalid (=0), AES XTS (=1), AES GCM (=2), AES CBC (=3), AES CTS (=4), AES HEH (=126), and private (=127). The nonce and the key descriptor are printed in hexadecimal format; the encryption modes are printed as human-readable literal from our istat extension.

For the extension of content and name decryption, we had to adjust several points in the TSK code. Fig. 3 shows which functions we had modified (in red) or added (in blue). The function `tsk_fs_fls` is exemplary for a call where filenames are outputted. The functions `tsk_fs_attr_read` and `tsk_fs_attr_walk` are the starting point when file content should be extracted.

In the linking process, the OpenSSL library is linked to the TSK executables to be able to perform decryption tasks. In the file `tsk/util/crypto_ext.c`, we implemented the decryption modes AES XTS and CTS. For XTS we only implemented a wrapper for the OpenSSL function. CTS is not provided by OpenSSL, therefore we implemented it by using AES ECB.

We added a new argument *-K* to the TSK command line tools *fls*, *fcat*, *icat*, and *ifind* which allows to add master keys to the root EXT4 filesystem structure in TSK (`EXT2FS_INFO`). With this argument, the user points to a file containing one master key per line in the format id:key_part1:key_part2.

TSK parses the EXT4 metadata structure and copies important data to a generic struct `TSK_FS_META`. We extended this generic structure to possibly point to encryption data. In the case of EXT4 filesystem this pointer points to a struct of type `ext2fs_encryption_attribute`. Future filesystems can use this generic pointer to reference their specific encryption attributes. Dependent on this change we had to adjust the filesystem independent TSK functions `tsk_fs_meta_reset` and `tsk_fs_meta_close` in `tsk/fs/fs_inode.c`. The functions `ext2fs_dinode_copy_encryption_attribute` and `ext2fs_extract_encryption_attribute` are responsible for initializing the encryption attributes, and are used in function `ext2fs_dinode_copy`.

To decrypt filenames and file content, we need to compute the DEK_f for every file or folder with encrypted content. The master keys of the filesystem are needed for key derivation. We store the master keys in the `EXT2FS_MASTER_KEY` structure, which can link multiple master keys together, and a reference to the first key is stored in the `EXT2FS_INFO` structure, which is the root data structure used in most TSK functions to store general filesystem information. `tsk_ext2fs_add_master_keys` adds the master keys to the EXT info structure. `tsk_fs_add_keys` is the general function which calls a file system specific function.

The key derivation of EXT4 is done in the function `ext2fs_calculate_file_key` which takes the metadata of a file and the `EXT2FS_INFO` structure. Directory entries, which store the filenames in EXT are parsed in the function `ext2fs_dent_parse_block` and `ext2fs_dent_copy`. We extended these functions to decrypt encrypted filenames.

If a user wants to read file content data with TSK command line tools or the library, in the end the functions `tsk_fs_attr_read` and `tsk_fs_attr_walk` are used to perform these tasks. The walk function calls a given callback function for every data chunk of a file. The read function can be used to extract a data chunk of a given size and offset. We extended these functions so that they call our special functions `ext4_walk_encrypted_data` and `ext4_read_encrypted_data` if the metadata shows that the

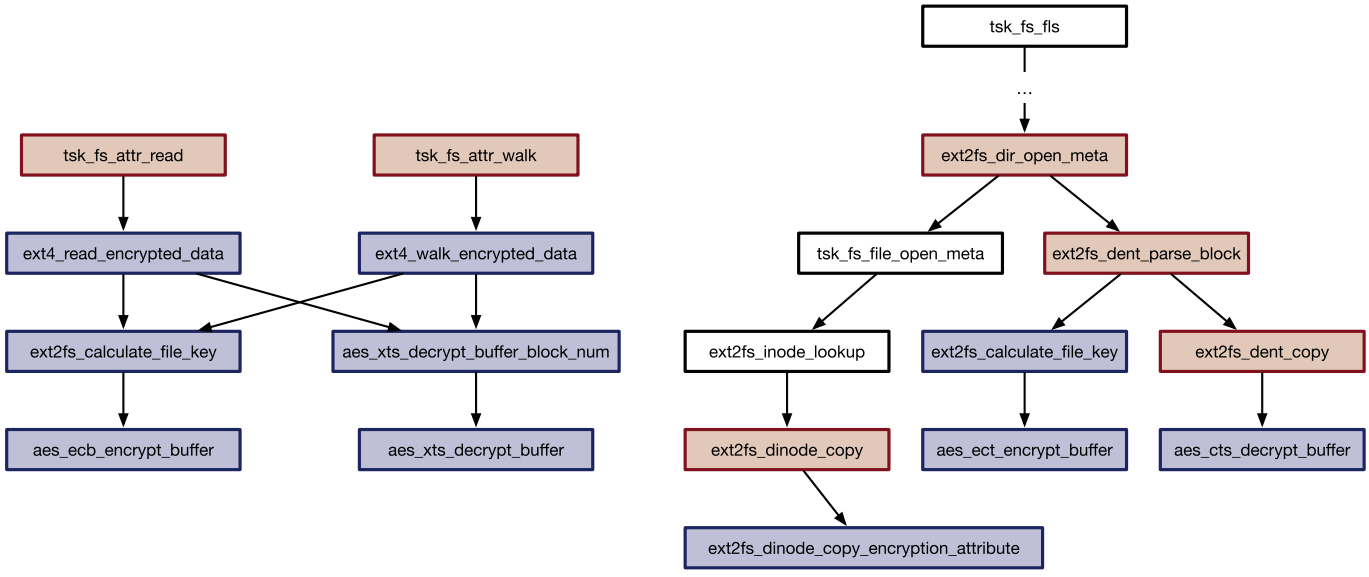

Fig. 3. Overview of functions of The Sleuth Kit (TSK), including unmodified functions (white), new functions (blue), and modified functions by us (red).

content is encrypted.

4.2. Fbekeyrecover

This tool takes a memory dump and an image of the filesystem to compute the used master keys. It uses the tools *aeskeyfind*, *fls*, and *istat* which get executed in an external process. With *fls*, all inodes present on the given EXT4 filesystem are collected. For every inode, *istat* is called to get the encryption metadata if present.

Of most importance to the tool are the nonce and the master key descriptor of an inode. All nonces are grouped by the master key descriptor. After this process, we have different bins of nonces.

The tool can process multiple memory dumps. This feature allows us to circumvent the crash of *aeskeyfind*, which occurs when processing too big memory dumps, by splitting the dumps into multiple files. Each memory dump gets processed by *aeskeyfind*, and all found 256-bit AES keys are collected in one list.

The last step of this tool is to compute all potential master keys per nonce bin. It computes the master keys in parallel by using eight workers. Thereby, it counts the occurrence of every master key. In the end, the keys are sorted by the count descending. The two 256-bit keys per nonce bin with the most hits shape the master key for the descriptor.

With our approach, only two file keys DEK_F are needed to be present on the memory dump to reconstruct a master key. This is valuable in cases where parts of the memory are already vanished or wiped during a cold boot attack.

4.3. Plaso extension

The extension of Plaso consists of three parts. First, we had to adjust the *pytsk3* project, which provided Python bindings to the TSK library. The second part is the extension of the dfVFS project, which provides data access on different abstraction levels (e.g., image, filesystem, files) by a path specification. This project uses the TSK library to open EXT filesystems. The last part is the extension of the Plaso project to allow EXT4 master keys to be provided by an argument to the command-line tools.

The *pytsk3* module provides the `FS_Info` object, which can be used to get file content or to list directory entries. We extended `FS_Info` with the `add_keys` method. It calls the `tsk_fs_add_keys` function of TSK, which introduces EXT4 master keys to TSK. Additionally, we had to link the library libcrypto to the pytsk3 module so that TSK is able to decrypt the data.

In the dfVFS project we added a `master_keys` attribute to the `TSKPathSpec` class. This path spec is responsible for such filesystems, which are handled by TSK (in our case EXT4). In dfVFS, different path specs can be linked together to create references to specific files or content in a file. Different path specs dependent on the items format are implement in the dfVFS project.

We modified the `_open` method of `TSKFileSystem` which is responsible for opening filesystem images. We added code to add the master keys specified in the path spec to the `FS_info` object from pytsk.

Since every TSK path spec should propagate the master keys to child TSK path specs we had to adjust the following methods: `_EntriesGenerator` of class `TSKDirectory`, `Get_Linked_File_Entry` and `GetParentFileEntry` of class `TSKFileEntry`, and `GetRootFileEntry` of class `TSKFileSystem`.

Lastly, we had to fix the serialization of path specs. During scanning through an image, Plaso serializes and deserializes path specs. To keep the master keys and to be able to propagate them to child TSK path specs we had to add `master_keys` to `PROPERTY_NAMES` in the class `Factory` (`dfvfs/path/factory.py`). This makes sure that master keys will be serialized and deserialized properly together with the path specs.

The Plaso tool already has an argument *−credential* that is used to provide f.e. BitLocker Drive Encryption passwords. We added a new credential type *ext4_master*. The credential argument is used in the format *ext4_master: <id>,<key_part1>,<key_part2>*.

One of the first calls of the Plaso command line tools *log2timeline* and *psteal* is `ScanSource` of the `StorageMedia Tool` class. This method opens the given source and searches for which partition and filesystem handler to use. If TSK is used because of an EXT filesystem, we add the master keys to the path spec with a call to our method `_addMasterKeys`.

Table 1
The devices from Google we did a full master key recovery for.

Release	Device	OS Version	Content Enc.	Name Enc.
2015	Google Nexus 5X	8.1.0	AES XTS	AES CBC CTS
2016	Google Pixel XL	10.0.0	private	AES CBC CTS
(2019)	Virtual Device	10	AES XTS	AES CBC CTS

5. Evaluation

We evaluated the applicability of our attack in two different ways. For two Google devices, we carried out a full master key recovery. Those devices are the Nexus 5X and the Pixel XL. For many other Android devices, we checked which encryption scheme is used, e.g., by extracting the encryption metadata, to draw a conclusion on the applicability of our attack. This includes devices from a wide range of vendors, such as Samsung, Huawei, and Xiaomi.

The devices where we successfully derived the master keys from file-specific data encryption keys present in a memory dump are shown in Table 1. For the Nexus 5X and the Pixel XL, we dumped the memory with the help of the *LiME* kernel module.[4] To be able to load this module and to dump the memory, we had compiled our own Android kernel from the sources. We enabled the loadable kernel module (LKM) support and compiled the *LiME* module together with the kernel.

With the tools *unmkbootimg* and *mkbootimg*[5] we inserted our custom kernel to an *boot.img* from a official factory image. Afterwards we flashed the device with the modified *boot.img* to get LKM support. In addition to the custom kernel, we rooted the devices with Magisk.[6]

By loading *LiME* we dumped the memory to an image. With the tools *dd* and *netcat*, we dumped the userdata partition as the root user to an image. Afterward, we called our master key recovery tool with both the memory and the userdata dump and evaluated the potential master keys. For each potential master key, we counted how often it was computed. For the Nexus 5X, the result can be seen in Fig. 4. On the x-axis the four most counted key candidates are displayed. As explained in Section 3, we expected to count only the real master keys multiple times. With this result, we can confirm this assumption.

On the Nexus 5X, we can observe that three different master keys are used. For every master key descriptor, we recovered two 256-bit keys multiple times. Since one master key is of size 512-bit (two keys of the size 256-bit), our attack is effective. If we consider the quantity, we can recover the complete 512-bit key in the right

[4] https://github.com/504ensicsLabs/LiME.
[5] https://github.com/difcareer/BootImgTool.
[6] https://magiskmanager.com.

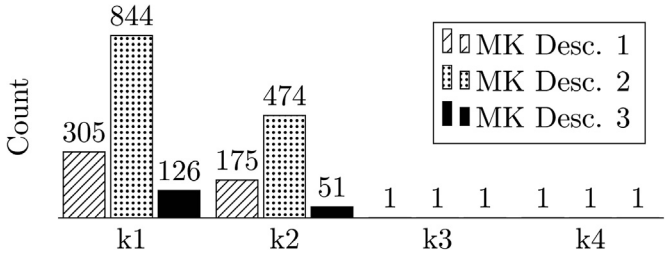

Fig. 4. Count of the same master key candidates for the Nexus 5X.

order. The key part found most times is the first part, the other one the second part.

We were also able to recover the master keys from an Android Virtual Device (AVD). Since Android 10, the AVD also uses FBE instead of FDE. We dumped the memory with the *qemu monitor*, and the userdata image was already present in the *qcow2* format.

During our evaluation of the three devices, we successfully decrypted filenames and content using the restored master keys and our modified TSK version. We evaluated the *Plaso* extension by reconstructing *Chrome* browsing events from the userdata image.

In the second part, we evaluate which disk encryption mechanism (e.g., FDE, FBE, FBE + metadata encryption) is used by smartphones we had access to. If a device uses FBE or metadata encryption, we also evaluated if it uses the name encryption mode, which indicates the usage of the KDF where we can recover master keys. This is the case for all name encryption modes except AES HEH.

This part of the evaluation is based on an Android application we implemented, which evaluates the outputs of the command *mount* and a bundled native binary. This binary reads the encryption metadata of the own data folder with the help of an `EXT4_IOC_GET_ENCRYPTION_POLICY` request via ioctl.

In Table 2, all devices we investigated are listed. Three of them, namely Nexus 6P, Mi 8, and P20 lite, still use plain FDE without metadata encryption. The name encryption mode indicates that they use the old KDF where we can recover the master keys. The Pixel 2 was the only device, which uses the *HEH* name encryption mode indicating its usage of a new KDF.

But our method can still be a building block in fully decrypting a metadata encrypted filesystem if you can recover the DEK used in FDE. Typically, this FDE key is not derived by a user's passphrase or pin. Instead, it is bound to the device hardware (typically in the TrustZone). If one can access this DEK, our method is still valuable for metadata encrypted devices to decrypt the FBE layer. The evaluation shows that the devices Pixel 3 and Pixel 4 use the metadata encryption together with FBE and a name encryption mode indicating the use of the old KDF. For the devices Galaxy S10 and P40 Pro, we had no access to the encryption metadata because the OS denied access and, therefore, we could not further evaluate the modes in use. This is displayed as ✳ in Table 2.

6. Conclusion

In this work, we have shown how to extract encryption keys for FBE, the encryption solution of choice for new smartphones, and mandatory for all devices running Android 10 or later. Our extraction method implemented in the tool *fbekeyrecover* differs notably from the extraction of FDE keys shown in previous work (Halderman et al., 2008). In contrast to FDE, where only one key is used to decrypt the whole partition, FBE uses one key per file, which results in hundreds of keys being held in RAM. We exploit the redundancy of key material in our method to make it robust against a partial wipe or loss of data from RAM because it is likely that the minimum of two needed file encryption keys is still present in the RAM after a cold boot attack. Previous work has shown that bit flips, caused by the remanence effect of DRAM, can destroy portions of RAM (Müller and Spreitzenbarth, 2013; Gruhn and Müller, 2013), possibly including file keys.

On memory dumps of two devices, namely the Google Nexus 5X and Pixel XL, we have shown that our method to recover FBE master keys works as intended. Although we were not able to perform all steps of a cold boot chain, due to the platform security features of modern Android phones, we still argue that our key deriving method is an important building block for law enforcement to successfully break FBE-enabled Android smartphones.

In the future, law enforcement has to use additional methods, such as malicious bootloaders and exploits, as we explained in Sect. 3.1.1, for example, to be able to obtain memory images from encrypted smartphones. Currently, the application of memory forensics moves from classic cold boot attacks, where devices could simply be rebooted or their RAM modules simply be transplanted, towards more sophisticated attack chains that also require breaking the platform security of a device, e.g., by attacking its secure boot scheme. More generally, to obtain a memory image via logical access (that is not by transplanting RAM modules), it is necessary to gain code execution at an appropriate level on the target phone. Since we found that file keys are stored in the memory space of the kernel, and *not* in more protected memory areas like the TrustZone, code execution at EL1 would suffice to have access to the keys we use.

One way to get such code execution is the mentioned EDL programmer, which is normally used by manufacturers as a last resort to repair a bricked phone. But in the past, we have also seen unintended flaws in boot ROMs that can be exploited to get code execution. For example, the *checkm8* exploit (Checkra1n Jailbreak: Anal, 2020) for iOS devices and a severe vulnerability affecting the Samsung Galaxy series (exynos-usbdl: unsigned co, 2020).

7. Limitation

In our study, we came across a new key derivation function (KDF) that is used for the implementation of FBE in the Android sources. As our recovery method was highly optimized for the old KDF, the new KDF renders our tool *fbekeyrecover* ineffective. But, the new KDF we investigated is only used if file names are encrypted with the AES HEH mode and already shipped devices will not be updated. Over-the-air updates do not change the encryption scheme of a device, because it requires a complete re-encryption of the userdata partition. As a consequence, for all devices where our *fbekeyrecover* tool is applicable today, it remains applicable in the future, too.

It can also be seen as a limitation that with the introduction of metadata encryption, FDE provides an additional layer of security against the type of attacks we presented. However, first, the FDE key is vulnerable to the exact same attacker model as the FBE key. Meaning that in scenarios where the FBE key can be recovered from RAM with our method, the FDE key can also be recovered with previously published methods (Halderman et al., 2008; Müller and Spreitzenbarth, 2013), if the key is not stored in a dedicated crypto element. Second, we argue that every encryption layer should be implemented properly on its own to protect data best. For example, in the past, there were flaws in the implementation of FDE (Busch et al., 2020), where the same static encryption key was used on many devices, making it easy for an attacker to decrypt the data. In this case, the correct implementation of FBE would indeed add security and protect user data despite FDE being flawed.

Table 2
Popular smartphones and their encryption schemes being used.

Release	Device	OS Version	Content Enc.	Name Enc.	old KDF	Metadata Enc.
2015	Samsung Galaxy S6	7.0	Full-Disk Encryption		–	–
2015	Google Nexus 6P	8.1.0	AES XTS	AES CBC CTS	✓	✗
2016	Huawei P9 lite	7.0	Full-Disk Encryption		–	–
2017	Google Pixel 2	10	private	AES HEH	✗	✗
2017	BQ Aquaris X	8.1.0	Full-Disk Encryption		–	–
2018	Google Pixel 3	9	private	AES CBC CTS	✓	✓
2018	Xiaomi Mi 8	8.1.0	private	AES CBC CTS	✓	✗
2018	Huawei P20 lite	8.0.0	AES XTS	AES CBC CTS	✓	✗
2019	Google Pixel 4	10	private	AES CBC CTS	✓	✓
2019	Samsung Galaxy S10	10	✳	✳	✳	✗
2020	Huawei P40 Pro	10.1.0	✳	✳	✳	(✓)

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) as part of the FIDI project. Special appreciation to our project partners, especially Prof. Dr. Konrad Rick, Dr. Daniel Arp and Robert Michael.

References

Bauer, J., Gruhn, M., Freiling, F.C., 2016. Lest we forget: cold-boot attacks on scrambled DDR3 memory. Digit. Invest. 16, S65–S74.

Busch, M., Westphal, J., Müller, T., 2020. Unearthing the TrustedCore: a critical review on huawei's trusted execution environment. In: Yarom, Y., Zennou, S. (Eds.), 14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020. USENIX Association. https://www.usenix.org/conference/woot20/presentation/busch.

Carrier, B., 2020. The Sleuth Kit. https://www.sleuthkit.org/sleuthkit/. (Accessed 27 September 2020).

Checkra1n Jailbreak: analysis of Checkm8 exploit. https://checkm8.info/blog/checkra1n-jailbreak-exploit, 2020, 01.10.2020.

Dent, A.W., 2020. Secure Boot and Image Authentication. https://www.qualcomm.com/media/documents/files/secure-boot-and-image-authentication-technical-overview-v2-0.pdf. (Accessed 10 July 2019).

Digital forensics virtual file system (dfVFS). https://plaso.readthedocs.io/en/latest/index.html, 2020–. (Accessed 27 September 2020).

Etemadieh, A., Heres, C., Hoang, K., 2017. Hacking hardware with $10 SD card reader. https://bh2017:exploitee:rs/Hacking_Hardware_With_A_10_Reader-wp:pdf blackhat 2017.

Exynos-Usbdl: Unsigned Code Loader for Exynos BootROM, 2020. https://fredericb.info/2020/06/exynos-usbdl-unsigned-code-loader-for-exynos-bootrom.html#exynos-usbdl-unsigned-code-loader-for-exynos-bootrom, 01.10.2020.

Garmany, B., Müller, T., 2013. PRIME: private RSA infrastructure for memory-less encryption. In: Proceedings of the 29th Annual Computer Security Applications Conference, ACSAC '13, Association for Computing Machinery, New York, NY, USA, December 9-13, 2013. ACM, pp. 149–158. https://doi.org/10.1145/2523649.2523656.

Götzfried, J., Müller, T., 2014. Analysing Android's full disk encryption feature. J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl. 5 (1), 84–100. https://doi.org/10.22667/JOWUA.2014.03.31.084, 10.22667/JOWUA.2014.03.31.084.

Götzfried, J., Müller, T., Drescher, G., Nürnberger, S., Backes, M., 2016a. RamCrypt: kernel-based address space encryption for user-mode processes. In: Chen, X., Wang, X., Huang, X. (Eds.), Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016. ACM, pp. 919–924. https://doi.org/10.1145/2897845.2897924, 10.1145/2897845.2897924.

Götzfried, J., Dorr, N., Palutke, R., Müller, T., 2016b. HyperCrypt: hypervisor-based encryption of kernel and user space. In: 11th International Conference on Availability, Reliability and Security, ARES 2016, Salzburg, Austria, August 31 - September 2, 2016. IEEE Computer Society, pp. 79–87. https://doi.org/10.1109/ARES.2016.13, 10.1109/ARES.2016.13.

Groß, T., Ahmadova, M., Müller, T., 2019. Analyzing Android's file-based encryption: information leakage through unencrypted metadata, 7. In: Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019, vol. 47. ACM, pp. 1–47. https://doi.org/10.1145/3339252.3340340, 10.1145/3339252.3340340.

Gruhn, M., Müller, T., 2013. On the practicability of cold boot attacks. In: 2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013. IEEE Computer Society, pp. 390–397. https://doi.org/10.1109/ARES.2013.52, 10.1109/ARES.2013.52.

Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W., 2008. Lest we remember: cold boot attacks on encryption keys. In: van Oorschot, P.C. (Ed.), Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008. USENIX Association, San Jose, CA, USA, pp. 45–60. http://www:usenix:org/events/sec08/tech/full_papers/halderman/halderman:pdf.

Hay, R., Hadad, N., 2018a. Exploiting qualcomm EDL programmers (1): gaining access & PBL internals. https://alephsecurity.com/2018/01/22/qualcomm-edl-1/. (Accessed 27 September 2020).

Hay, R., Hadad, N., 2018b. Exploiting Qualcomm EDL Programmers (2): Storage-Based Attacks & Rooting. https://alephsecurity.com/2018/01/22/qualcomm-edl-2/. (Accessed 27 September 2020).

Hilgers, C., Macht, H., Müller, T., Spreitzenbarth, M., 2014. Post-mortem memory analysis of cold-booted android devices. In: Freiling, F.C., Morgenstern, H., Frings, S., Göbel, O., Günther, D., Nedon, J., Schadt, D. (Eds.), Eighth International Conference on IT Security Incident Management & IT Forensics, IMF 2014, Münster, Germany, May 12-14, 2014. IEEE Computer Society, pp. 62–75. https://doi.org/10.1109/IMF.2014.8, 10.1109/IMF.2014.8.

hisi-idt, 2020. https://github.com/96boards/burn-boot, 01.10.2020.

Ligh, M.H., 2014. TrueCrypt Master Key Extraction and Volume Identification. https://volatility-labs.blogspot.com/2014/01/truecrypt-master-key-extraction-and.html, 01.10.2020.

Loftus, R., Baumann, M., van Galen, R., de Vries, R., 2017. Android 7 File Based Encryption and the Attacks against it. University of Amsterdam, p. 33.

Metadata Encryption, 2020. https://source.android.com/security/encryption/metadata. (Accessed 27 September 2020).

Mikhaylov, I., 2016. Chip-off technique in mobile forensics. https://www.digitalforensics.com/blog/chip-off-technique-in-mobile-forensics/. (Accessed 27 September 2020).

Müller, T., Freiling, F.C., Dewald, A., 2011. TRESOR runs encryption securely outside RAM. In: 20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011. USENIX Association. Proceedings. http://static:usenix:org/events/sec11/tech/full_papers/Muller:pdf.

Müller, T., Spreitzenbarth, M., 2013. Frost - forensic recovery of scrambled telephones. In: J Jr., M.J., Locasto, M.E., Mohassel, P., Safavi-Naini, R. (Eds.), Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings, Vol. 7954 of *Lecture Notes In Computer Science*. Springer, pp. 373–388. https://doi.org/10.1007/978-3-642-38980-1_23.

Redini, N., Machiry, A., Das, D., Fratantonio, Y., Bianchi, A., Gustafson, E., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2013. Bootstomp: on the security of bootloaders in mobile devices. In: 26th USENIX Security Symposium USENIX Security 17, pp. 781–798.

Sammons, J., 2012. The Basics of Digital Forensics: the Primer for Getting Started in Digital Forensics. Elsevier.

Skillen, A., Barrera, D., van Oorschot, P.C., 2013. Deadbolt: locking down android disk encryption. In: Enck, W., Felt, A.P., Asokan, N. (Eds.), SPSM'13, Proceedings of the 2013 ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2013, November 8, 2013, Berlin, Germany. ACM, pp. 3–14. https://doi.org/10.1145/2516760.2516771, 10.1145/2516760.2516771.

Teufl, P., Fitzek, A., Hein, D.M., Marsalek, A., Oprisnik, A., Zefferer, T., 2014. Android encryption systems. In: 2014 International Conference on Privacy and Security in Mobile Systems, PRISMS 2014, Aalborg, Denmark, May 11-14, 2014. IEEE, pp. 1–8. https://doi.org/10.1109/PRISMS.2014.6970599, 10.1109/PRISMS.2014.6970599.

Unterluggauer, T., Mangard, S., 2016. Exploiting the physical disparity: side-channel attacks on memory encryption. In: Standaert, F., Oswald, E. (Eds.), Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers, Vol. 9689 of *Lecture Notes In Computer Science*. Springer, pp. 3–18. https://doi.org/10.1007/978-3-319-43283-0_1, 10.1007/978-3-319-43283-0_1.

Wang, Z., Murmuria, R., Stavrou, A., 2012. Implementing and optimizing an encryption filesystem on android. In: Aberer, K., Joshi, A., Mukherjea, S., Chakraborty, D., Lu, H., Venkatasubramanian, N., Kanhere, S.S. (Eds.), 13th IEEE International Conference on Mobile Data Management, MDM 2012, Bengaluru, India, July 23-26, 2012. IEEE Computer Society, pp. 52–62. https://doi.org/10.1109/MDM.2012.31, 10.1109/MDM.2012.31.

Welcome to the Plaso documentation. https://plaso.readthedocs.io/en/latest/index.

html, 2020—. (Accessed 27 September 2020).
Würstlein, A., Gernoth, M., Götzfried, J., Müller, T., 2016. Exzess: hardware-based RAM encryption against physical memory disclosure. In: Hannig, F., Cardoso, J.M.P., Pionteck, T., Fey, D., Schröder-Preikschat, W., Teich, J. (Eds.), Architecture of Computing Systems - ARCS 2016 - 29th International Conference, Nuremberg, Germany, April 4-7, 2016, Proceedings, Vol. 9637 of *Lecture Notes In Computer Science*. Springer, pp. 60—71. https://doi.org/10.1007/978-3-319-30695-7_5, 10.1007/978-3-319-30695-7_5.