TU WIEN Informatics

Analysis of Android Factory Resets

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software und Information Engineering

eingereicht von

Michael Lang
Matrikelnummer 01328838

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Mitwirkung: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik
Univ.Lektor Dipl.-Ing. Christian Kudera
Dipl.-Ing. Michael Pucher

Wien, 22. April 2022

Michael Lang	Edgar Weippl

TU WIEN Informatics

Analysis of Android Factory Resets

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Michael Lang
Registration Number 01328838

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik
Univ.Lektor Dipl.-Ing. Christian Kudera
Dipl.-Ing. Michael Pucher

Vienna, 22nd April, 2022

Michael Lang | Edgar Weippl

Erklärung zur Verfassung der Arbeit

Michael Lang

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. April 2022

Michael Lang

Abstract

Smartphones are our every day companions. We use them to communicate, perform payments, track activities, or perform work-related tasks. With 2.5 billion Android devices in circulation, it's not surprising that there also exists a substantial second hand market for smartphones. For example, in 2017 18% of people opted to sell their old device.

The fact that private or company data resides on smartphones is a potential risk to privacy or confidentiality when selling or discarding an old phone. Because phyisical destruction is not an option if the phone is to be sold, the only viable alternative is logical sanitization. For this purpose, Android provides the factory reset functionality, which is supposed to delete all personal data.

In this paper we take a look at the factory reset implementation of Android from versions 5 to 9. We look at how the flash memory is wiped and which concrete operations are performed. We identify major changes between the versions and document them. Additionally, we look at the factory reset implementations of three popular alternatives whose factory reset implementation is based on Androids, namely LineageOS, OxygenOS and KaiOS.

We found no apparent issues in Android versions 5 to 8. We found that in Android 9 a change to the factory reset implementation causes the use of the non-secure *ioctl(BLKDISCARD)* instead of it's secure counterpart *BLKSECDISCARD*. We also identified some minor issues in the other operating systems.

Contents

CHAPTER 1

Introduction

This chapter contains the motivation and problem statement. We talk about current smartphone usage habits and about the second-hand smartphone market, which leads to the problem statement of securely wiping a phone. Finally, this chapter contains contributions and the structure of this work.

1.1 Motivation

Smartphone capabilities grew a lot; from the first phone that included a full-fledged browser in 2007 to personal assistants that manage appointments and make reservations in 2020. These new capabilities changed the way people utilize phones. Phones allow us to check our emails, make financial transactions, keep track of our diet or manage our commute. In addition to personal information they may also contain company data: based on a survey from 2018 around 45% of people in the UK use their smartphones for accessing work mail and about 30% for keeping track of appointments [1].

The fact that private and/or company data resides on smartphones can be a potential risk to privacy or confidentiality, for example when one wants to sell their phone. In 2017, about 18% of people who replaced their device opted to resell their old one, while around 35% kept it as a spare [2]. A survey from 2017 suggests that people replace their phones on average after 21 months [3]. Even if a phone is not re-used, the improper disposal still comes with security implications. To avoid that others gain access to private information, it is paramount to securely wipe the memory of the phone.

Nowadays, Android is the most used mobile OS with 2.5 billion devices as of 2019 [4]. With the success of Android, multiple open source OSes based on Android emerged. One example is LineageOS[1], a fork of the Android Open Source Project (AOSP) which supports 150 different devices and has over 1.7 million active installations [5].

[1] `https://lineageos.org/`

Android provides the factory reset feature, which claims to delete all user data from the device [6].

Factory reset is also the recommendation of NIST SP 800-88 *Guidelines for Media Sanitization* [7] to dispose data of an Android device with the caveat that the OEM correctly implemented the storage drivers and the storage provides a secure erase function.

Previous works analysed factory reset for Android 2.3 to 4.3. To our knowledge, no studies about the effectiveness of factory resets for newer versions of Android exist.

1.2 Problem Statement

As discussed in Section 1.1, a phone generally contains sensitive data, be it phone numbers, private photos or company data.

The different stakeholders want to keep this data confidential. Relatives and friends don't want their phone numbers leaked, employers want to protect information about new projects and the user wants to keep their medical records private.

This presents the challenge of securely deleting the data when selling or disposing the phone. One way to ensure that the data is not recoverable is to physically destroy the device. This option, although the most secure when done the right way, is mainly suitable for companies which can afford a sanitization device such as a shredder. Physical destruction also does not apply when the phone is to be sold or reused inside the company. A more feasible alternative, especially for individuals, is logical sanitization of the device.

This cannot be done manually by the user, because deleting a file via an file explorer most often just deletes the reference to that file, but actually keeps the contents on the flash memory. Additionally, deleting every file manually would be very time consuming and could lead to errors, e.g. users missing certain directories. A better solution is to use the factory reset functionality provided by Android, which clears the userdata and cache partitions by performing a secure delete operation based on the underlying file system / flash storage. This ensures that the data is really deleted from the storage. For details about flash memory and secure deletion see Section 3.2.

Earlier versions of Android had issues with Factory Resets, which made it possible to recover private data, see Chapter 2. Therefore, it is paramount that the factory reset operation works correctly.

1.3 Contribution and Methodology

In this work, we will take a look at the implementation details of the factory reset mechanisms to find out if these issues still persist in Android 5 to 9. Additionally, we look at three other mobile operating systems, namely LineageOS [2] and OxygenOS [3], both of

[2] https://lineageos.org/

[3] https://github.com/OnePlusOSS

which are forks of Android, and KaiOS [4], which is based on the Boot to Gecko operating system. We examine how these systems implement factory reset, if the implementations differ from the Android Open Source Project and if they carried over old flaws of the Android implementation.

To achieve this, we will examine the source code of Android and the other operating systems, compare the relevant code and document relevant differences.

1.4 Structure of this Work

Chapter 1 introduces the topic to the reader. Chapter 2 gives an overview of the existing work covering factory reset and other closely related works. The necessary background for the remaining chapters is presented in Chapter 3. Chapter 4 discusses the implementation details of the factory reset functionality across the different versions of the respective operating systems. Chapter 5 presents the results of the analysis from Chapter 3. Finally, Chapter 6 summarises the findings and concludes the work.

[4] https://www.kaiostech.com/

CHAPTER 2

Related Work

In chapter 2 we briefly touch upon related topics. In particular, we provide existing works about the Android factory reset, secure deletion of storage media in general, and forensic data recovery for flash memory and Android devices.

2.1 Factory Reset

There already exists work on the security of built-in methods for resetting mobile devices to it's factory state.

Schwamm et al. [8] performed an empirical study on factory resets for Android, iOS and Blackberry devices. They compared pre- and post-reset artefacts and were able to recover data on both devices. They did not identify why the data was not properly sanitized.

A more thorough analysis of the factory reset functionality of Android 2.3 to 4.3 was performed by Simon et al. [9]. They were able to recover data across all versions and identified two causes, the use of the *ioctl BLKDISCARD* syscall and missing driver support for secure sanitisation methods of the flash memory chip.

Shu et al. [10] looked data deletion of Android, including the recovery system, and possible flaws leading to residual data. They also discovered flaws in the factory reset method of third-party or custom recovery systems.

Because modern mobile devices exclusively use flash memory as persistent storage media, this topic is closely related to secure deletion methods for flash memory.

2.2 Secure Deletion

Secure deletion of storage media is a widely researched topic [11],[12],[13].

Based on the type of storage medium, e.g. magnetic or optical storage, different methods have to be used to securely remove the data [7], [12].

For flash storage, this proves to be especially hard due to the fact that data cannot be updated in-place due to the discrepancy in granularity between the program and erase operations, see chapter 3.2.

To overcome this problem, a range of secure deletion methods were proposed.

One possibility is to use cryptographic erasure [7], i.e. encrypting the data and erasing the encryption key. This method was initially proposed by Boneh et al. [14] to simultaneously delete files from the file system and accompanying backups.

Lee et al. [15] developed a flash file system based on YAFFS2[1] that supports secure deletion using cryptographic erasure.

A user space file system for Android that provides secure deletion was proposed and implemented by Yang et al. [16]. They ensure deletion of the encryption keys by completely filling the storage medium and therefore force garbage collection of unmapped blocks.

Another possibility is to provide an operation in the flash translation layer that securely deletes data.

An evaluation of garbage collection times of deleted blocks and differences in response time between compulsory garbage collection after each delete and normal operation was performed by Kwak et al. [17].

To overcome the disadvantages of the naive way of secure deletion in the FTL, Wei et al. proposed an operation they call *scrubbing*, that is reprogramming an already programmed page and setting all bits, effectively erasing the data [18].

Jia et al. introduced a technique they call *NAND flash partial scrubbing*, which improves on the method proposed by Wei et al. [18] by also being undetectable by an adversary [19].

In the case of mobile devices, secure deletion, together with encryption, is an important piece in keeping mobile phone data confidential by preventing the recovery of data of stolen or second-hand phones.

2.3 Forensic Methods

In the case of flaws in the factory reset functionality some or all user data that should have been deleted could be left on the flash chip. This section lists research about mobile data acquisition methods that could be used to retrieve that data.

For mobile devices, there are different types of acquisition methods. They are generally categorized into logical and physical acquisition [20],[21],[22].

[1] https://yaffs.net

The most common physical acquisition methods are JTAG, ISP and chip-off. Physical acquisition methods are typically invasive, more difficult to perform and require specialised hardware, but have the benefits of bypassing screen-locks and of obtaining a physical image of the chip.

Logical or software based acquisition methods include among others the Android Debug Bridge (adb) or commercial forensics software that utilizes a combination of different methods, for example using root exploits to extract data.

Surveys of different acquisition methods were performed by Scrivens et al. [20] and Sathe et al. [22].

Yang et al. reverse engineered firmware update protocols for some Android devices and used remnant read commands to acquire an image of the flash memory [23].

Hasan et al. use a chip-off method to recover data after it was deleted via *scrubbing* [24].

CHAPTER 3

Background

In this chapter we provide the background needed to understand the discussion in chapter 4. We given an overview of the Android OS and talk about the recovery image, the boot process and security mechanisms of Android. Then we briefly discuss current standards of flash storage and their capabilities.

3.1 Android

Android is an open source operating system by Google which was revealed in 2007. It is based on the long time support (LTS) release of the Linux kernel with some Android-specific changes from the AOSP and from OEMs. These changes are primarily to improve power consumption, to support mobile hardware or other file systems or improve security [25].

On top of the kernel sits the Hardware Abstraction Layer (HAL). The HAL is a set of interfaces which are implemented by the OEM or SoC manufacturer and provide access to the device-specific hardware to the higher layers of the Android system. This allows the Android framework to be independent from the underlying hardware and has the advantage that system updates do not require the vendors to rebuild the HAL.

From Android 7.1 onward, Android supports seamless updates using an A/B-partition scheme. Up to and including Android 10, seamless updates are optional, meaning device manufacturer could choose if the device is an A/B device or not.

A/B devices have two slots, namely slot A and slot B, of the partitions boot, vendor, radio and system. When an update is available, it is applied to the inactive partitions, e.g. slot B. When the update is finished, the device is rebooted from slot B and the slot A partitions are marked as inactive. To support A/B updates, devices require a different partitioning scheme than non-A/B devices. In the following, we give an overview of the

partitions in Android and highlight changes between different versions and between A/B and non-A/B devices [26].

- **boot** contains the linux kernel image. Non-A/B devices with Android 8 or lower also contain a ramdisk with a bootable rootfs. A/B devices instead contain a recovery ramdisk which, in addition to the boot code, contains the code for the recovery mode.
- **system** contains the Android framework and for A/B devices and Android ≥ 9 also a rootfs.
- **user / userdata** contains user-installed apps and customization data.
- **cache** stores temporary data for caching and is optional if a device implements A/B updates.
- **metadata** is used to store metadata for device encryption.
- **vendor** hold vendor specific code and customisations. In older versions of Android, OEMs typically applied their code and customisation directly to the Android source, mixing vendor specific and generic code. With Android 8, all vendor specific code is now separated from the Android source [27]. This enables Android updates without the need for OEMs to modify their source code.
- **recovery** stores the recovery image, i.e. a linux kernel and the recovery mode. This partition does not exist on A/B devices. For A/B devices, updates are applied by the *update_engine* module of the currently unused system partition slot and the recovery mode itself is located in a ramdisk in the boot partition.
- **misc** is used by the bootloader and recovery to communicate with eachother. Contains the bootloader control block (BCB) [28].
- **sdcard** is the primary shared storage. It contains files received via file transfer or media files.

These differences in the partitioning between the Android versions and A/B and non-A/B devices mainly affect the concrete boot and update process. For example, Android 9 or later will directly mount the system partition as the root filesystem (at /) (which is called system-as-root) and start the first stage init process from the system partition, where previous versions would mount the ramdisk in the boot partition and start init from there.

Android supports the ext4 and f2fs file systems for block-based storage, i.e. embedded MultiMediaCard (eMMC) or Universal Flash Storage (UFS) and YAFFS2 for raw nand storage [26]. Nowadays, devices typically use eMMC or the newer UFS storage. We will further discuss flash storage in chapter 3.2

For the factory reset functionality, relevant partitions are boot, data / userdata, cache, metadata, recovery and misc.

3.1.1 Bootloader

The bootloader is responsible for loading either the kernel or the recovery image into memory and starting it. Additionaly, the bootloader may implement Verified Boot. Verified Boot was introduced with Android 4.4. It is a specification for verifying the integrity and authenticity of the installed system. If an manufacturer decides to implement Verified Boot, they sign the built version of Android on their devices and embed the public part of the key in read-only storage. During boot, the bootloader then verifies the signature on the partitions using this public key. With Android 8 a reference implementation of Verified Boot called Android Verified Boot is provided, which adds rollback protection. Rollback protection prevents flashing of older images over newer ones by comparing their version.

A device can be locked or unlocked; the bootloader prevents booting of an unsigned OS only if the device is in the locked state. With an unlocked device, it is possible to install a custom recovery system or kernel or a modified version of Android. Unlocking is also a popular method to gain root access to a device. This can be achieved by unlocking the bootloader and flashing a custom *boot* image which enables apps to run with root permissions, for example see [29]. Because unlocking a device disables the security model of Android, some manufacturers disable the unlock functionality by setting the system property *oem_unlock_supported* to false before building the *system* image.

The bootloader decides whether to load the recovery image or the kernel by reading the command field in the bootloader control block (BCB) in the *misc* partition. This value is written to by Android when, for example, the user initializes a factory reset from the settings app.

3.1.2 Recovery

The purpose of the recovery mode is to apply over-the-air (OTA) updates and to allow users to reset the device. Recovery consists of a kernel and the code for the recovery mode, i.e. the recovery binary. The binary contains among other things a minimalistic GUI, the code for the factory reset and update functionalities and code for communicating with the bootloader.

For non-A/B devices, the recovery is self-sufficient, all the code is located in the recovery partition. For A/B devices, the recovery mode (code/data) is in the recovery ramdisk in the boot partition, but the kernel is reused from the inactive boot slot.

The recovery binary is controlled by arguments from three different sources. These are in decreasing priority the command line, the bootloader control block (BCB), and the file /cache/recovery/command. When the binary is started, it parses it's arguments and performs the appropriate action.

3.1.3 Security Mechanisms

Because we carry mobile devices with us most of the time, they are easy targets for theft. For example, in the UK alone, in 2015 about 538.000 people were victims of mobile phone theft [30].

To prevent personal data from being stolen, Android 4.4 introduced full-disk encryption (FDE). Android 5 made it mandatory for devices to support FDE, but it was not required by default. With Android 6, encryption has to be enabled by default for devices that have an AES encryption performance above 50 MiB/s [31].

For FDE, a random 128-bit disk encryption key (DEK) is generated and encrypted using the user credentials, i.e. the PIN, password or pattern or the string 'default_password' and a hardware-bound key stored in the trusted execution environment (TEE) of the device. The DEK is then used to encrypt the data partition.

FDE has the disadvantage that the phone cannot be used without the user entering their credentials. To eliminate this disadvantage, file-based encryption (FBE) was released with Android 7. In contrast to full-disk encryption, file-based encryption uses two keys, a device encrypted (DE) and a credential encrypted (CE) key. Both keys are encrypted using a key stored in the TEE, the CE key in addition is encrypted using the user credentials. This allows apps to access storage encrypted with the DE key without the need for the user to enter their credentials. The actual encryption of the user data is done by assigning an encryption policy to each directory in the data partition. This policy specifies if the directory is encrypted using the DE or CE key. In addition to the actual files the filenames are also encrypted.

Up to Android 9, devices must support at least one encryption method and enable it by default, with Google recommending file-based over full-disk encryption.

For a detailed description about encryption see [32].

Altough encryption guarantees the integrity of user data, which is good from a privacy perspective, it certainly does not deter criminals from stealing phones. Before Android 5, a stolen phone which was protected by a PIN or passphrase could still be used by booting into the recovery mode via a certain keystroke combination and performing a factory reset. This would clear the userdata, cache and metadata partitions and effectively return the phone to it's initial state after leaving the factory, hence the name factory reset. This is not entirely true because the system partition is not reset, meaning updates will remain.

To discourage theft, the factory reset protection (FRP) feature was introduced with Android 5.1. FRP requires that the credentials of the previous owner of the device are entered during the setup process if the devices was reset using the recovery mode. This is done by using a separate partition, typically /dev/block/.../by-name/frp, which stores the required data in a persistent data block (PDB). When a PDB is present during the device setup process, the previous user's credentials are checked. If the check fails, the device will not boot. Legitimate users can wipe the PDB by initialising a factory

reset from the device settings UI or removing all Google accounts from the device. An adversary cannot do that, even if they have access to the unlocked device, because both actions require the credentials of the user.

Android protects apps and their data by using existing features of the Linux kernel among others. Each application is run in a separate process and is assigned a unique user ID, which prevents apps from accessing resources which they do not have permission for. Since Android 5 SELinux is used to further limit the access to other processes and files. Users and apps by default do not have root permissions in Android.

It is important that the bootloader is locked, see chapter 3.1.1. With an unlocked bootloader, everyone with physical access to the device can overwrite the device partitions with custom software and bypass all security mechanisms.

3.2 Flash Storage

Mobile devices use flash storage for their non-volatile memory. The two most common types of flash storage design are NOR and NAND flash [33]. For the purpose of mass storage NAND flash is generally used due to it's much higher storage density and lower cost [34, Chapter 6.1].

The information about flash memory in the following paragraph is based on [34].

NAND memory is structured in blocks, which consist of multiple pages which in turn consist of cells. A cell can store one or more bits of information. Cells which can store one bit are called single-level cells (SLC) and cells that can store more bits multi-level cells (MLC). MLC memory has the advantage that it has a higher storage density than SLC memory, but a shorter lifespan. The size of a page is typically in the range of a kilobyte, e.g. 2 KB, whereas blocks typically are in the range of 32 or more pages. A block additionally stores some metadata, i.e. the number of program / erase cycles, the block state and/or an error correction code. Raw flash memory has the operations erase, program/write and read. The program and read operations are performed on a page, whereas an erase operation targets a whole block [34, Chapter 6.1]. NAND memory does not allow random access, i.e. it is not possible to update already written pages; pages have to be deleted before they can be programmed again. The cells of memory degrade with each erase operation. The lifespan of the cells is typically measured in write/erase cycles, that is the minimal number of write/erase operations which can reliably be performed on each block, as guaranteed by the manufacturer. This typically is in the range of 10.000 for MLC and 100.000 for SLC memory.

Because of these limitations of NAND flash, additonal memory management is needed. When a file is written or updated, the memory pages cannot be updated in place. Instead they have to be erased or copied before being written again. This decreases the write performance due to additional erase and program operations and wears out some blocks faster, which in turn decreases the lifetime of the NAND storage overall. This is especially bad for mobile devices which have an embedded memory. This problem is typically solved

by using either managed flash, which has a built-in controller with a flash translation layer (FTL) [35], or by using a dedicated flash file system [36].

Both FTL and the flash file system perform the additional memory management operations and offer in-place update functionality to the rest of the system. To ensure the maximal lifetime of the flash, all blocks should be used evenly. FTLs typically implement a wear leveling algorithm [37] which tries to achieve that.

Another technique to increase the overall lifetime is to over-provision the memory [38], i.e. the flash contains a percentage more blocks than advertised and when a block is marked as bad it is replaced by one of the extra blocks.

Early smartphones used to ship with unmanaged or raw NAND flash and therefore had to use a flash file system, e.g. YAFFS2. Now, only managed flash is used for mobile devices. The two main standards for managed flash are eMMC [39] and it's successor UFS [40].

Both eMMC and UFS memory have an integrated controller who manages the actual low level access to the flash.

3.2.1 Secure Deletion

As discussed above, flash requires wear leveling which in case of managed NAND is performed by the memory controller. When a delete operation is performed, the controller may not actually delete the blocks but rather mark them as unused. This is not always the desired behaviour, for example when we want to delete sensitive data or perform a factory reset of an Android phone.

For this usecase, the eMMC standard specifies the *sanitize* or *secure erase* operations respectively. The *sanitize* operation succeeds the *secure erase* operation with eMMC version 4.51.

To securely delete data, the *erase* operation is executed first. *Erase* moves all given blocks from the mapped to the unmapped address range. Then, the *sanitize* operation has to be executed; this physically removes all data from the unmapped address space. The concrete operation that is performed by *sanitize* depends on the Secure Removal Type byte. Four different actions are supported [39]:

1. Erasing of the physical memory.
2. Overwriting the addressed locations with a character followed by an erase.
3. Overwriting the addressed locations with a character, its complement, then a random character.
4. Using a vendor defined function.

For example an eMMC 5.0 module from Toshiba supports method 1 and 4. It is important to notice that the available erase operations depend on the vendor.

Linux provides access to the functions of the eMMC flash via the ioctl syscall. The command for the block device driver is *BLKSECDISCARD*, but the flash can also be securely erased by issuing a *MMC_IOC_CMD* ioctl with the appropriate parameters.

The difference between the two is that *BLKSECDISCARD* ioctl issues a *secure trim* or *secure erase* operation, and with *MMC_IOC_CMD* it is possible to issue every supported command, e.g. *sanitize*.

The eMMC *secure erase* and *secure trim* operations are supported in the Linux kernel since August 2010 [41], and the *sanitize* operation since May 2013 [42].

CHAPTER 4

Analysis

In this chapter we examine the implementations of the factory reset functionality in Android, LineageOS, KaiOS and OxygenOS. We begin by taking a detailed look at the factory reset code in Android, as it is the basis of the other implementations. After that, we introduce the implementations of the other operating systems by comparing them to Android.

4.1 Android Open Source Project

This section contains the analysis of the Android factory reset implementation. In short, a factory reset first prepares the bootloader control block (BCB) and reboots the device into the recovery system. There, the *format_volume* function reformats the volumes */data*, */cache* and */metadata*. Then the BCB is reset and the device again reboots into the main system.

The following sections take a deep dive into the concrete implementations of each step.

4.1.1 Triggering a Factory Reset

In Android, there are three ways for the user to start a factory reset; directly from the recovery system, from the settings app or via the remote administration functionality of Android / Google.

Additionally, factory reset is also triggered every time the bootloader is unlocked, see 3.1.1.

When the user starts a factory reset via settings or the remote administration functionality, an *ACTION_FACTORY_RESET* intent is broadcasted. This triggers the *MasterClearReceiver.onReceive* action which in turn calls *RecoverySystem.rebootWipeUserData(..)*. *reboootWipeUserData* writes the arguments for recovery into the BCB. They are *–wipe_data*,

–reason=MasterClearConfirm and *–locale=default-locale*. Then, the system is rebooted. During the restart, the bootloader reads it's arguments from the BCB and then boots into recovery [43].

On the other hand, a factory reset can be triggered by selecting the corresponding option, typically *wipe data/factory reset*, in the recovery system. This does not write to the BCB or reboot the device but instead directly performs a factory reset. A device can be booted into recovery mode by either using the Android Debug Bridge (adb) with the *reboot recovery* command [44] or by pressing certain keys while the phone is starting, e.g. power, volume up and home for Samsung devices.

4.1.2 Inside the Recovery System

The recovery system first parses it's arguments from one of the following three sources in decreasing priority; the actual command line, the bootloader control block (BCB) or the contents of the recovery command file, usually */cache/recovery/command* [45]. In the case of a factory reset triggered by the user the commands are supplied via the BCB. They are, as mentioned above, *wipe_data*, *locale* and *reason*. The value of *reason* and *locale* is logged and *locale* is used to initiate the UI with the correct language. Then follows a long *if ... else if* statement that performs different actions based on the supplied arguments. In the case of *wipe_data*, the function *wipe_data* is called. This function is the single entry point for a factory reset. All relevant operations regarding a factory reset are performed by this function and it's subroutines. It performs five major steps: *pre-wipe*, *wipe /data*, *wipe /cache*, *wipe /metadata*, and *post-wipe*.

The pre- and post-wipe subroutines perform device-specific operations and are supplied by the vendor. For example, the post-wipe routine in the Pixel 3 phones clears data from the Titan M security module. The default implementation provided by Android does not perform any actions.

In the case that the user directly booted into recovery, a menu is shown that allows the user to select an action. If *wipe data* is selected, the *wipe_data* function is called directly.

4.1.3 Wiping volumes

The function that wipes the volumes is called *erase_volume*. It takes the volume path as an argument. As mentioned above, it is called with the values */data* and, if the volumes exist, with */cache* and */metadata*. The function *erase_volume* first initializes the UI background and displays a progress bar. If the volume to wipe is the *cache* volume, log files which are stored there are temporarily loaded into memory and restored again after the formatting is done. That is done to prevent deleting the logs of the current run of recovery.

Then, the *format_volume* function is called. The function *format_volume* first initialises a struct *v* of type fstab_rec using the volume path. The struct contains the data of the entry for the volume in the recovery fstab file. Then some checks are performed, e.g. if

the given path exists and is a valid volume, the given path is not the ramdisk, and the volume is not mounted. Furthermore, factory reset is only supported for *ext4* or *f2fs* filesystems. If any of these checks fail, the factory reset will be terminated.

Next, the device encryption metadata is wiped, if it exists. The path to the encryption metadata is read from one of the flags *forcefdeorfbe*, *encryptable*, or *forceencrypt* in the *recovery.fstab* file and stored in *v->key_loc*. The wiping is done using the *wipe_block_device* function from the *system/extras/ext4_utils* module [46]. In *wipe_block_device*, it is first checked if the device is really a block device. As mentioned in chapter 3, block devices do not allow raw access to the disk but rather hide the hardware-specific details behind an integrated memory controller, e.g. for managing wear leveling. Therefore, the wipe instruction is only needed for block devices.

The secure deletion is done by issuing a *BLKSECDISCARD* operation via the *ioctl* system call. The range of blocks that should be erased is given as an argument and is always $[0, n]$ with n being the total size of the device. In case the operation fails, e.g. the underlying controller does not support the *BLKSECDISCARD* operation, the command *BLKDISCARD* is used as a fallback mechanism. If that fails too, a warning message is shown, or, since Android 10, the device is manually overwritten with zeroes. The fallback mechanisms only performs logical sanitization of the blocks, i.e. the blocks are only marked as deleted but the data is still present. This may allow recovery of the underlying data as shown in e.g. [9], [18].

Recreating the file system

The next step is recreating the files system. For *f2fs*, this is done using *mkfs.f2fs* from f2fs-tools [47]. By default, *mkfs.f2fs* trims the device before recreating the file system. This is done by issuing the *BLKSECDISCARD* command to the device via the *ioctl* syscall. If that fails, the *BLKDISCARD* command is used as a fallback mechanism. In Android 8 and lower, the default trim operation is disabled via the *-t* option. From Android 9 onwards, the *-t* option is no longer used and therefore the device is securely wiped.

For *ext4fs*, there are two different code bases. Up to Android Oreo, the file system is created by calling *make_ext4fs_directory* from the *system/extras/ext4_utils* module. This in turn calls *wipe_block_device* to securely wipe the partition [46].

Since Android 9, the *mke2fs* binary from *e2fsprogs* is used to create the file system [48]. This change was introduced in revision *ded2dac* [49]. *mke2fs* also supports discarding the device before file system creation, but it uses *BLKDISCARD* to do so [50]. As mentioned above, *BLKDISCARD* does not guarantee that blocks are physically wiped and it may allow recovery of the underlying data. We suspect this change is a bug due to the fact that the *wipe_block_device* function in *wipe.cpp* is still maintained [46].

4.1.4 Cleanup

After the file system creation the recovery system arguments, in this case the BCB, are reset and the system is rebooted.

4.1.5 Summary

In summary, a factory reset first boots into the recovery system. The volumes */data*, */cache* and */metadata* are reformated using either the *BLKSECDISCARD* or *BLKDISCARD* operation. If one of the volumes is encrypted, the associated encryption metadata is always wiped using *BLKSECDISCARD*. Then, the file system is recreated, parameters are cleaned up and the system is restarted. Because the */system* partition is not cleared during a factory reset, operating system updates are not reset.

Android versions from Lollipop to Oreo (5 to 8.1) use the *ioctl(BLKSECDISCARD)* syscall to securely clear the user data.

From version Pie (9) and upwards, *mke2fs* is used to create the file system. Here, *ioctl(BLKDISCARD)* is used to wipe the partition, which may allow recovery of the data [9], [18].

4.1.6 System Call Handling

As stated above, Android uses the *BLKSECDISCARD* or *BLKDISCARD* ioctl system call to wipe the flash memory when a factory reset is performed. These system calls are handled by the kernel and the device driver. Therefore, the security of the wipe operation depends on the concrete driver and product kernel of the device.

In 2017, Google wanted to unify the kernel landscape of Android devices by introducing the Android Common Kernels (ACKs). ACKs are downstream of long term releases of the Linux kernel and the base for all device kernels [51]. They contain Android specific features, patches, and drivers and tools common to all Android devices. Google also introduced a kernel version requirement; since Android 8 Oreo, new Android devices have to launch with a kernel versions from a designated list [25]. Older devices may still use older versions of the Linux kernel, even if they are upgraded to Android 8 or higher.

Next, we will look at how the *BLKSECDISCARD* iotcl operation is handled in the kernel. We choose to examine ACKs 3.18, 4.4, 4.9. 4.14 and 4.19, because they contain all designated release kernels for Android versions 8.0 to 10. In addition we examined kernel 3.10, because it is used by KaiOS which we cover in chapter 4.4.

In all mentioned kernel versions, the *ioctl* syscall is handled by the generic block layer. First, in */block/ioctl.c* a switch statement over the *ioctl* operation decides which operation to call. In the case of *BLKSECDISCARD* the function *blkdev_issue_discard* from */block/blk-lib.c* gets called. This function prepares the block request in form of a linked list of *bio* structures, which is then handed to the specific device driver by calling *submit_bio*. For more information about Linux drivers see [52] or [53].

The request is then handled by the MultiMediaCard (MMC) driver code in */drivers/mmc/*. First, the range of blocks that should be erased is selected. This is done by issuing the commands *ERASE_GROUP_START* and *ERASE_GROUP_END* with the start and end addresses as 32 bit argument to the eMMC controller. As mentioned above, in case of a factory reset this range spans the whole devices. Then the *ERASE* command with argument 0x80000000 is issued, which corresponds to a secure erase as defined by the eMMC standard [39].

For all Android Common Kernels we looked at, the MMC driver uses the above mentioned requests.

Since eMMC 4.5, the recommended way to securely delete the data is the *SANITIZE* operation [39]. *SANITIZE* has to be used after an *ERASE* or *TRIM* operation and physically removes the data from the unmapped address space.

Support for the *SANITIZE* operation was added to the kernel in October 2011. It was used in the *mmc_blk_issue_secdiscard_rq* function of the MMC driver, which handles the *BLKSECDISCARD* operation. At first, the *SANITIZE* operation was called without a prior *ERASE* or *TRIM* operation. This meant that the respective blocks were not deleted at all, and only the unmapped address space was wiped. This bug was fixed in April 2012 and applied in kernel version 3.2.

Interestingly, the use of the *SANITIZE* operation in the *BLKSECDISCARD* handler of the MMC driver was reverted in April 2013 and instead moved to the *MMC_IOC_CMD* ioctl syscall. The *MMC_IOC_CMD* operation is the pass-through command for MMC devices, i.e. it allows sending of arbitrary commands to MMC devices. Since then, the MMC driver again uses the *ERASE* or *TRIM* operation with the secure erase arguments. The relevant commit message states that the *SANITIZE* operation was removed because it's purpose is to be invoked by a user and not by the file system or via a block device request.

Altough the security of the wipe operation should not be affected by using either *ERASE* and *SANITIZE*, or the secure variant of the *ERASE* command, the wipe operation in Android should use the recommended way to securely remove the data from flash devices. Using the deprecated functionality may introduce bugs or unexpected behaviour in some device configurations.

4.2 LineageOS

LineageOS, formerly known as CyanogenMod, is a custom ROM based on Android. The ROM is currently available for about 170 different devices [54].

The source code of LineageOS is forked from the AOSP project and then modified. The ROM developers also maintain the device tree and device kernel with the drivers for each specific hardware. Table 4.1 shows the version of LineageOS with the corresponding AOSP version.

For our analysis, only the recovery system, bootable/recovery, and the system/extras module are of interest. We compared these modules with their upstream Android projects by adding a second remote to the git project, e.g. `https://android.googlesource.com/platform/bootable/recovery` for the recovery module, and using the *Compare with Branch* feature of Android Studio.

Version 13, 14 and 14.1 include preprocessor directives which change the behaviour of *wipe_block_device* based on the existence of macros.

If *NO_SECURE_DISCARD* is defined, only *BLKDISCARD* is used to sanitize the device, if *SUPPRESS_EMMC_WIPE* is defined, sanitizing is skipped all together and only a warning is displayed [55]. These macros are set based on the board-specific configuration in *BoardConfig.mk* of the device during the build.

The versions 15, 15.1, 16, 17, and 17.1 of LineageOS are functionally identical to their respective Android versions regarding the factory reset functionality. LineageOS 15 and 15.1 only have some minor adjustments in the code responsible for displaying GUI elements, e.g. the progress bar. Some interesting changes introduced by LineageOS are that from version 16 and upwards, it is possible to install unsigned updates by skipping the verification step. Version 17 adds the option to wipe the system partition to the recovery menu [56].

The kernels for concrete devices, e.g. for Xiaomi or OnePlus devices, are based on the Android Common Kernel Project or on the *Android for MSM Project* kernels [57]. The *Android for MSM Project* kernels are downstream of ACKs and include qualcomm chipset support. Therefore, it is safe to assume that the LineageOS kernels handle the system calls in the same way as the ACKs.

CM 13	refs/tags/android-6.0.1_r81
CM 14	refs/tags/android-7.0.0_r14
CM 14.1	refs/tags/android-7.1.2_r36
LineageOS 15	refs/tags/android-8.0.0_r30
LineageOS 15.1	refs/tags/android-8.1.0_r52
LineageOS 16	refs/tags/android-9.0.0_r46
LineageOS 17	refs/tags/android-10.0.0_r11
LineageOS 17.1	refs/tags/android-10.0.0_r41

Table 4.1: LineageOS versions with corresponding versions of Android

4.3 OxygenOS

OxygenOS is a custom ROM developed by OnePlus Technology for their One Plus devices.

In 2016, the source code for OxygenOS was made public on Github [58]. Both the kernel and the Android source are based on the Android for MSM project. These

repos are downstream of AOSP and Android Common Kernels respectively and contain enhancements for Qualcomm Snapdragen SoC.

OnePlus publishes the source code in form of a repo manifest file. This file includes all the repositories needed to build the specific android image.

For our analysis only the */system/extras* module and the recovery system are of importance. As with LineageOS, we performed the analysis by getting the manifest files for different versions of OxygenOS [58]. We then cloned the */system/extras* and the */bootable/recovery* repositories in the version specified in the manifest file. The comparison was done by using the *Compare with Branch* feature of Android Studio using the corresponding upstream branch of AOSP. Table 4.2 shows the versions of OxygenOS with their respective Android versions.

All versions of the OxygenOS recovery were identical to the AOSP implementation regarding the factory reset functionality. The *wipe_block_device* function in *system/extras/ext4_utils/wipe.c* was also identical for all versions.

OxygenOS 3.5	Android 6.0.1 (marshmallow-release)
OxygenOS 4.5	Android 7.1.2 (nougat-mr2-release)
OxygenOS 5.0	Android 8.0 (oreo-release)
OxygenOS 5.1	Android 8.1 (oreo-m8-release)
OxygenOS 9	Android 9 (pie-release)

Table 4.2: OxygenOS versions with corresponding versions of Android

4.4 KaiOS

KaiOS is an operating system for feature phones. It is a fork of *Boot to Gecko* (B2G) [59]. Gecko is the browser engine developed by Mozilla. In case of B2G and KaiOS, Gecko acts as the application runtime, i.e. it is the counterpart to Android's *Android Runtime (ART)*. Therefore, apps for KaiOS are written using web technologies, i.e. HTML, JS, CSS.

To allow Gecko access to hardware functionality via web APIs, it is built with Gonk as a porting target. Gonk is the name for the underlying operating system. It consists of an AOSP Linux kernel, device libraries and drivers, and a hardware abstraction layer [60].

B2G is licensed under MPL[1], therefore the parts of KaiOS that are based on B2G are also made available on Github [61]. For the analysis we also looked at the publicly available parts of the source code of the Nokia 8110 4G, which runs KaiOS [62]. Unfortunately, the UI layer and the system apps are not open source. Therefore our analysis of KaiOS only covers Gecko, Gonk, and the recovery image.

[1]Mozialla Public License `https://www.mozilla.org/en-US/MPL/2.0/`

KaiOS generally provides three ways to reset the phone: via the settings app, from the recovery system or via remote wiping provided by the KaiOS anti-theft service [2]. The available options for a specific model depend on the vendor.

KaiOS uses the stock AOSP recovery image. In the repo manifest file we see that revision *refs/tags/android-10.0.0_r1* from `https://android.googlesource.com` is referenced. KaiOS does not state which version of the Linux kernel they use. The kernel version used by the Nokia 8110 4G is 3.10.

To interact with the recovery system, the *librecovery* library is used. It provides the functions to start a factory reset or an *FOTA* (firmware-over-the-air) update. This is done by writing the desired command to the recovery command file accordingly and then rebooting into the recovery system. For a factory reset the command *–wipe_data* is used. For more information on interacting with the recovery system also see chapter 4.1.2.

The functionality of *librecovery* is exposed to Gecko via the *RecoveryService.js* module and the *js-ctypes* library. *js-ctypes* is a wrapper for shared libraries that allows calling C/C++ functions from JavaScript code [63]. *RecoveryService.js* provides the *factoryReset(reason)* method. Based on the argument, it writes different commands to the post reset command file and then starts a factory reset. As is the case with Android, the factory reset wipes the volumes */data* and, if they exist, */cache* and */metadata*. If the value *wipe* is passed to the *factoryReset(..)* method, potential SD card partitions are also cleared during the first start up after the factory reset. This is done by adding all partitions in the post reset command file and, after rebooting, recursively deleting all files and subdirectories in the partitions. The files are deleted using the *nsILocalFile* interface. For Unix systems, the implementation can be found in *xpcom/io/nsLocalFileUnix.cpp*. There we see that *unlink* is used to delete a file. Therefore, there are no guarantees that the data on the SD card was securely removed from the flash memory.

Because the recovery system used is identical to the AOSP recovery, all observations from chapter 4.1 also apply to KaiOS. In particular the usage of *e2fsprogs*' *mke2fs* to recreate the file system is noteworthy. As discussed in chapter 4.1, *mke2fs* replaced the functionality of the *system/extras/ext4_utils* module since Android 9. Therefore, *wipe_block_device* and in turn the *ioctl(BLKSECDISCARD)* syscall are no longer used to wipe the partitions. Instead, *mke2fs* wipes the partitions using the *ioctl(BLKDISCARD)* syscall. This does not guarantee that the unmapped address space of the flash memory is deleted.

[2] `https://services.kaiostech.com/antitheft/#/login`

CHAPTER 5

Results

Chapter 5 summarizes our findings from chapter 4. We discuss the issues affecting Android and highlight notable changes introduced by the other operating systems.

5.1 Overview

The security of the factory reset of a phone depends on the fact that every single component out of the flash device, the kernel as well as the operating system support secure deletion.

As discussed in chapter 3.2, the standard for managed flash memory used in smartphones is either eMMC or it's successor UFS. The eMMC standard supports secure deletion since version 4.5, i.e. since 2012, with the *secure erase* or *secure trim*, or the newer and current *sanitize* command [39]. For UFS flash memory, secure removal is mandatory since at least version UFS 2.0, i.e. since 2013 [64].

Secondly, the kernel has to support the eMMC or UFS standard as mentioned above. Support for the eMMC *secure erase* operation is provided by the block device driver of the kernel [41]. To trigger the secure erase, an *ioctl(BLKSECDISCARD)* system call can be used. In May 2013 support for eMMC version 4.5 was added to the Linux kernel. Since then, the *ioct(MMC_IOC_CMD)* system call can be used to issue the sanitize command to a flash device [42].

Lastly, the OS has to use the correct system call to trigger a secure deletion operation. In the case of a factory reset, the Android recovery image contains the code responsible for securely wiping the device.

5.2 Findings

As shown by Simon et al. [9], early versions of Android had problems with sanitisation during a factory reset. Even though Android 4.0 and onwards uses the secure *ioctl(BLKSECDISCARD)* operation to sanitize the data partition, some devices they analysed did not support the *BLKSECDISCARD* command. The suspected that vendors did not include the necessary device drivers to support secure deletion. They also showed that with increased versions, the devices with insecure deletions decreased.

With that said, from Android 5 to 8 there where no significant modifications in the factory reset code, most importantly the *ioctl(BLKSECDISCARD)* is universally used for sanitization.

In Android 9, with revision *ded2dac* [49], the *mke2fs* binary from the *e2fsprogs* module replaces the call to the *make_ext4fs_directory* function from the *system/extras/ext4_utils* module. With this change, the function *wipe_block_device* is no longer called during a factory reset, see chapter 4.1.2. Although *mke2fs* supports wiping the device before recreating the file system, it does so by issuing a *ioctl(BLKDISCARD)* operation. In contrast to the *ioctl(BLKSECDISCARD)* operation, this does not correspond to a secure deletion operation as defined by the eMMC [39] and UFS [40] standards.

The recovery system of LineageOS has only minor changes to the respective Android version. Therefore, starting from LineageOS 16, which corresponds to Android 9, the factory reset is also affected by the change to use *mke2fs* instead of *make_ext4fs_directory* as described above.

In LineageOS 13, 14 and 14.1 the behaviour of *wipe_block_device* can be adjusted during build. It is possible to skip the deletion of the flash device entirely or to downgrade to the use of *BLKDISCARD* [55]. This was done to prevent some devices to be permanently damaged due to bugs in the Samsung eMMC firmware. This was only relevant for older devices, e.g. Samsung Galaxy S2, but the code was kept until LineageOS 14.1.

The recovery system used by OxygenOS is identical to the respective Android versions. Therefore, from version 9, OxygenOS also suffers from the use of *BLKDISCARD* for wiping the flash memory.

Although KaiOS is based on *Boot to Gecko* rather than Android, it uses the Android recovery image to reset the device, see chapter 4.4. At the time of writing, the current version of KaiOS is 3.0 and uses the recovery image version *refs/tags/android-10.0.0_r1*. That means the observations for Android also apply and it may be possible to recover data after a factory reset.

In addition, KaiOS supports wiping of the SD card partition using the *unlink* system call, which does not have any security guarantees.

CHAPTER 6

Conclusion

In this work, we examined the factory reset mechanism of Android from version 5 to 9 and compared it with the implementations of some of the major Android-based ROMs LineageOS and OxygenOS, and with the successor of Boot to Gecko, KaiOS.

We looked at the source code for each major version of Android and identified differences between these versions. We identified a major change between Android 8.1 and Android 9 that changed the behaviour of the factory reset mechanism, specifically the way the file system is recreated during a factory reset. As a consequence of these changes, the way the sanitization of the flash memory is performed also changed. It no longer uses the secure *ioctl(BLKSECDISCARD)* system call but rather the non-secure version *ioctl(BLKDISCARD)*. As discussed in this work and in literature [24], [65], [66] this may retain data in the flash memory and allow for recovery of some or all of the data.

The other operating systems we looked at use their own recovery images, which are forks of the Android recovery. They only have little or no changes to the original source code. Commit *ded2dac* was also pulled into each code base, i.e. the issue described above is present in LineageOS, OxygenOS as well as KaiOS starting from the versions Android 9, LineageOS 16, OxygenOS 9, and KaiOS 3.

One possible way to fix this problem would be to call *wipe_block_device* during the factory reset as done in earlier Android versions.

Bibliography

[1] Deloitte, "Deloitte Global Mobile Consumer Survey UK edition." http://www.deloitte.co.uk/mobileuk2018/, June 2018. Accessed on 19.04.2020.

[2] Deloitte, "Global mobile consumer trends, 2nd edition." https://www2.deloitte.com/content/dam/Deloitte/global/Documents/Technology-Media-Telecommunications/gx-global-mobile-consumer-survey-2nd-edition.pdf, 2017. Accessed on 18.04.2020.

[3] T. Lu, "Smartphone Users Replace Their Device Every Twenty-One Months." https://www.counterpointresearch.com/smartphone-users-Dreplace-their-device%2every-twenty-one-months, October 2017. Accessed on 05.05.2020.

[4] "Google I/O 2019 Keynote." https://events.google.com/io2019/, 2019. Accessed on 18.08.2020.

[5] "LineageOS Statistics." https://www.lineageoslog.com/statistics, 2020. Accessed on 16.08.2020.

[6] "Reset your Android device to factory settings." https://support.google.com/android/answer/6088915?hl=en, 2020. Accessed on 18.08.2020.

[7] A. R. Regenscheid, L. Feldman, and G. A. Witte, "NIST Special Publication 800-88, Revision 1: Guidelines for Media Sanitization," tech. rep., National Institute of Standards and Technology, February 2015.

[8] R. Schwamm and N. C. Rowe, "Effects of the factory reset on mobile devices," *Journal of digital forensics, security and law*, vol. 9, no. 2, p. 17, 2014.

[9] L. Simon and R. Anderson, "Security Analysis of Android Factory Resets," in *Proceedings of 4th Workshop on Mobile Security Technologies (MoST)*, 2015.

[10] J. Shu, Y. Zhang, J. Li, B. Li, and D. Gu, "Why data deletion fails? a study on deletion flaws and data remanence in android systems," vol. 16, jan 2017.

[11] P. Gutmann, "Secure deletion of data from magnetic and solid-state memory," in *6th USENIX Security Symposium (USENIX Security 96)*, (San Jose, CA), pp. 77–89, 1996.

[12] S. M. Diesburg and A.-I. A. Wang, "A survey of confidential data storage and deletion methods," vol. 43, dec 2010.

[13] J. Reardon, D. Basin, and S. Capkun, "Sok: Secure data deletion," in *2013 IEEE symposium on security and privacy*, pp. 301–315, IEEE, 2013.

[14] D. Boneh and R. J. Lipton, "A revocable backup system.," in *6th USENIX Security Symposium (USENIX Security 96)*, (San Jose, CA), 1996.

[15] J. Lee, J. Heo, Y. Cho, J. Hong, and S. Y. Shin, "Secure deletion for nand flash file system," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, (New York, NY, USA), p. 1710–1714, Association for Computing Machinery, 2008.

[16] L. Yang, T. Wei, F. Zhang, and J. Ma, "Sadus: Secure data deletion in user space for mobile devices," *computers & security*, vol. 77, pp. 612–626, 2018.

[17] J. Kwak, H. C. Kim, I. H. Park, and Y. H. Song, "Anti-forensic deletion scheme for flash storage systems," in *2016 IEEE International Conference on Network Infrastructure and Digital Content (IC-NIDC)*, pp. 317–321, IEEE, 2016.

[18] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives.," in *FAST*, vol. 11, pp. 8–8, 2011.

[19] S. Jia, L. Xia, B. Chen, and P. Liu, "Nfps: Adding undetectable secure deletion to flash translation layer," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, (New York, NY, USA), p. 305–315, Association for Computing Machinery, 2016.

[20] N. Scrivens and X. Lin, "Android digital forensics: Data, extraction and analysis," ACM TUR-C '17, Association for Computing Machinery, 2017.

[21] S. Pappas, "Investigation of jtag and isp techniques for forensic procedures," Master's thesis, UNIVERSITY OF TARTU Institute of Computer Science, Ülikooli 18, 50090 Tartu, ESTONIA, 2017.

[22] S. C. Sathe and N. M. Dongre, "Data acquisition techniques in mobile forensics," in *2018 2nd International Conference on Inventive Systems and Control (ICISC)*, pp. 280–286, 2018.

[23] S. J. Yang, J. H. Choi, K. B. Kim, and T. Chang, "New acquisition method based on firmware update protocols for android smartphones," *Digital Investigation*, vol. 14, pp. S68–S76, 2015. The Proceedings of the Fifteenth Annual DFRWS Conference.

[24] M. M. Hasan and B. Ray, "Data recovery from "scrubbed" nand flash storage: Need for analog sanitiza-tion," 2020.

[25] "Android Common Kernels." https://source.android.com/devices/architecture/kernel/android-common. Accessed on 25.05.2020.

[26] "Android Architecture: Partitions and Images." https://source.android.com/devices/bootloader/partitions-images. Accessed on 28.04.2020.

[27] "Android Architecture." https://source.android.com/devices/architecture. Accessed on 20.08.2021.

[28] The Android Open Source Project, "Android source code, bootloader_message.h." https://android.googlesource.com/platform/bootable/recovery/+/refs/heads/master/bootloader_message/include/bootloader_message/bootloader_message.h. Accessed on 27.04.2020.

[29] J. Wu, "Magisk." https://github.com/topjohnwu/Magisk. Accessed on 12.05.2020.

[30] Home Office UK, "Reducing moblie phone theft and improving security." https://www.gov.uk/government/publications/reducing-mobile-phone-theft-and-improving-security-paper-2, 2016. Accessed on 01.05.2020.

[31] Google Inc., "Android compatibility definition document." https://source.android.com/compatibility/cdd. Accessed on 02.05.2020.

[32] "Android Security: Encryption." https://source.android.com/security/encryption. Accessed on 28.04.2020.

[33] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to flash memory," *Proceedings of the IEEE*, vol. 91, no. 4, pp. 489–502, 2003.

[34] J. Brewer and M. Gill, *Nonvolatile memory technologies with emphasis on flash: a comprehensive guide to understanding and using flash memory devices*, vol. 8. John Wiley & Sons, 2011.

[35] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for nand flash memory," in *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pp. 161–170, 2006.

[36] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system.," in *USENIX*, pp. 155–164, 1995.

[37] L.-P. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, (New York, NY, USA), p. 1126–1130, Association for Computing Machinery, 2007.

[38] Kingston Technology Corporation, "Understanding ssd over-provisioning (op)." https://www.kingston.com/austria/us/ssd/overprovisioning. Accessed on 20.09.2020.

[39] JEDEC Solid State Technology Association, "EMBEDDED MULTI-MEDIA CARD (e•MMC), ELECTRICAL STANDARD (5.1)," tech. rep., JEDEC Solid State Technology Association, 2019.

[40] JEDEC Solid State Technology Association, "UNIVERSAL FLASH STORAGE (UFS), Version 3.0," Tech. Rep. JESD220D, JEDEC Solid State Technology Association, January 2018.

[41] A. Hunter, "Linux kernel MMC driver." https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?&id=dfe86cba7676d58db8de7e623f5e72f1b0d3ca35. Accessed on 06.12.2020.

[42] M. Erez, "Linux kernel MMC driver." https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=775a9362b5d7e006ff6bbec5cb9c9c9d5a751696. Accessed on 06.12.2020.

[43] "Android Source Code." https://android.googlesource.com/platform/manifest/. Accessed on 19.04.2020.

[44] "Android Debug Bridge." https://android.googlesource.com/platform/packages/modules/adb/. Accessed on 19.03.2021.

[45] The Android Open Source Project, "Android recovery image." https://android.googlesource.com/platform/bootable/recovery/. Accessed on 19.03.2021.

[46] "Android ext4 utils Module." https://android.googlesource.com/platform/system/extras/+/refs/heads/master/ext4_utils/. Accessed on 05.05.2021.

[47] "Android f2fs-tools Source Code." https://android.googlesource.com/platform/external/f2fs-tools//. Accessed on 05.05.2021.

[48] "Android e2fsprog Source Code." https://android.googlesource.com/platform/external/e2fsprogs/. Accessed on 27.04.2021.

[49] "Android Source Code - Revision ded2dac." https://android.googlesource.com/platform/bootable/recovery/+/ded2dac082fd703f1cd7a5c3de59450cd3dc2530. Accessed on 24.04.2021.

[50] "e2fsprog Unix I/O Manager discard ." https://android.googlesource.com/platform/external/e2fsprogs/+/refs/heads/master/lib/ext2fs/unix_io.c#1084. Accessed on 02.07.2021.

[51] "Generic Kernel Image." https://source.android.com/devices/architecture/kernel/generic-kernel-image. Accessed on 25.12.2020.

[52] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, Third Edition*. O'Reilly, 2005.

[53] The kernel development community, "Kernel documentation." https://www.kernel.org/doc/html/latest//. Accessed on 20.12.2020.

[54] "LineageOS build targets." https://github.com/LineageOS/hudson/blob/master/lineage-build-targets. Accessed on 31.01.2021.

[55] "LineageOS wipe.c Source Code." https://github.com/LineageOS/android_system_extras/blob/cm-14.1/ext4_utils/wipe.c. Accessed on 31.06.2021.

[56] "LineageOS recovery system Source Code." https://github.com/LineageOS/android_bootable_recovery. Accessed on 31.06.2021.

[57] "Android for MSM Project." https://wiki.codeaurora.org/xwiki/bin/QAEP/. Accessed on 17.01.2021.

[58] OnePlus Technology (Shenzhen) Co., Ltd, "One plus open source software." https://github.com/OnePlusOSS. Accessed on 10.07.2021.

[59] "Kaios." https://developer.kaiostech.com/. Accessed on 27.10.2021.

[60] Mozilla Developer Network and individual contributors, "Firefox os platform - gonk." https://web.archive.org/web/20150906002012/https://developer.mozilla.org/en-US/Firefox_OS/Platform/Gonk. Accessed on 07.11.2021.

[61] KAI OS TECHNOLOGIES (HONG KONG) LIMITED, "Kaios open source software." https://github.com/kaiostech/. Accessed on 27.10.2021.

[62] HMD Global, "Nokia open source software." https://www.nokia.com/phones/en_int/opensource. Accessed on 27.10.2021.

[63] Mozilla Developer Network and individual contributors, "js-ctypes." https://web.archive.org/web/20190929211939/http://developer.mozilla.org/en-US/docs/Mozilla/js-ctypes. Accessed on 20.11.2021.

[64] JEDEC Solid State Technology Association, "UNIVERSAL FLASH STORAGE (UFS), Version 2.0," Tech. Rep. JESD220B, JEDEC Solid State Technology Association, September 2013.

[65] J. Luck and M. Stokes, "An integrated approach to recovering deleted files from nand flash data," 2008.

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