Analysing Android's Full Disk Encryption Feature

Johannes Götzfried* and Tilo Müller
Friedrich-Alexander-Universität
Erlangen-Nürnberg, Germany
{johannes.goetzfried, tilo.mueller}@cs.fau.de

Abstract

Since Android 4.0, which was released in October 2011, users of Android smartphones are provided with a built-in encryption feature to protect their home partitions. In the work at hand, we give a structured analysis of this software-based encryption solution. For example, software-based encryption always requires at least a small part of the disk to remain unencrypted; in Android this is the entire system partition. Unencrypted parts of a disk can be read out and are open to system manipulations. We present a tool named EvilDroid to show that with physical access to an encrypted smartphone only (i.e., without user level privileges), the Android system partition can be subverted with keylogging. Additionally, as it was exemplary shown by attacks against Galaxy Nexus devices in 2012, Android-driven ARM devices are vulnerable to cold boot attacks. Data recovery tools like FROST exploit the remanence effect of RAM to recover data from encrypted smartphones, at worst the disk encryption key. With a Linux kernel module named Armored, we demonstrate that Android's software encryption can be improved to withstand cold boot attacks by performing AES entirely on the CPU without RAM. As a consequence, cold boot attacks on encryption keys can be defeated. We present both a detailed security and a performance analysis of Armored.

Keywords: cold boot, evil maid, Android, cpu-bound encryption

1 Introduction

Smartphones are long since part of the privacy sphere of their owners, and consequently, a missing smartphone that falls into the wrong hands has severe consequences for its owner. Since most people use the same device for private and business applications [1], private as well as corporate data is often at risk of getting stolen. To protect mobile data on smartphones against physical theft, private people and company regulations started to switch from simple PIN locks to encryption technologies. Setting a PIN lock is not sufficient for protecting data against unauthorized access because adversaries can bypass the system software and access storage media directly, e.g., by replugging and accessing the storage memory directly.

Up to 2013, different technologies have been developed to enforce the *physical security layer* of smartphones for protecting the data on smartphones against physical theft. Among those are anti-theft solutions with "remote disable" and "remote wipe" functionalities. These features enable smartphone owners to deactivate or delete their sensitive contents remotely over a network connection. They are provided by virtually all anti-virus vendors for smartphones, including Kaspersky, F-Secure, and Lookout. Network connections, however, can simply be interrupted by adversaries by removing the SIM card, such that these countermeasures provide only limited security. Encryption solutions based on AES are considered much stronger than anti-theft solutions to prevent unauthorized access to data. With Android 4.0 (released in October 2011), Google introduced an AES-based encryption solution for smartphones. Android 3.0 (released in February 2011) also supports encryption, but this version was only available on a few tablet PCs and has never been widespread.

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, volume: 5, number: 1, pp. 84-100

*Corresponding author: Martensstr. 3, 91058 Erlangen, Germany, Tel: +49-9131-85-69904

1.1 Disk Encryption Security

With the support for encryption on Android devices, the question arises whether attacks on full disk encryption known from desktop computers like *cold boot attacks* or *evil maid attacks* can be performed on smartphones as well.

For example, every few years, researchers warn against something we often forget – the *remanence effect* of main memory [2, 3]. The remanence effect says that RAM is less volatile than most people expect because RAM contents fade away gradually over time rather than being lost immediately after power is cycled off. Attacks based on this effect have a long history beginning in 1996, when Anderson and Kuhn [4] proposed the first theoretic attack exploiting the remanence effect. Later in 2001, Gutmann [5] extended the basic idea by Anderson and Kuhn, and provided a detailed insight into the remanence effect. As a consequence of his insights, Gutmann suggested not to store cryptographic keys in RAM a long time ago. Nevertheless, people continued to store all data in RAM carefree, including disk encryption keys. In 2008, disk encryption keys were then successfully recovered from RAM by Halderman et al. [6]. For the first time, Halderman et al. were able to break disk encryption by exploiting the remanence effect.

Additionally, software-based disk encryption is known to be vulnerable to so-called *evil maid attacks*. On PCs, such attacks have become widely known in 2009 when Rutkowska [7] showed how to subvert bootloaders of encrypted hard drives with keylogging. Such attacks can, for example, be performed by "evil maids" gaining physical access to a laptop that was left unattended in a hotel room. Once the disk encryption password of a laptop is logged, it can be read out during a second physical access such that the hard drive can eventually be decrypted, effectively breaking disk encryption.

1.2 Contributions

We asked if attacks known from PCs also succeed against encrypted smartphones and – if so – which differences must be taken into account and which steps can be taken against these attacks. In a nutshell, we present two main contributions in this article, one being offensive and the other being defensive:

- With EvilDroid, we show that evil maid attacks can be deployed against smartphone owners, too. Since Android does not only leave the MBR unencrypted but also the entire system partition, evil maid attacks become perfectly possible.
- With Armored, we show an exemplary solution that hardens smartphones against cold boot attacks. To this end, we store necessary keys and intermediate values of AES inside CPU registers of the ARM microprocessor, i.e. without involving main memory.

We designed both EvilDroid and Armored for ARM devices with Android OS version 4.0 installed. Both prototype implementations were designed on a PandaBoard (an ARM development board) but we also tested them successfully on real smartphones like the Galaxy Nexus device from Samsung. Whereas EvilDroid is yet unpublished work, Armored was originally published at the ARES conference 2013 [8].

1.2.1 EvilDroid: Evil Maid goes after Android

Up to 2013, it has not been shown if and *how* the stock version of Android is vulnerable to evil maid attacks. With our tool named "EvilDroid" we basically support two attack modes:

- *Classic Evil Maid*: On smartphones with an unlocked bootloader, or where the bootloader can get unlocked without destroying user data, we subvert the system partition similar to MBRs and install EvilDroid directly to it. EvilDroid modifies the original PIN prompt of Android in a way that it additionally logs the PIN and stores it to the unencrypted cache partition. With a second physical access, we read out the PIN from the cache partition, and eventually access all data on the phone.

- *Networked Evil Maid*: If the bootloader of a target device is locked and cannot get unlocked without destroying the user partition, we cannot subvert the system partition. For such phones, we developed a second variant of EvilDroid where we exchange the target phone with an identical model. If a victim logs into the replaced phone, his or her PIN is immediately send to the attacker via SMS or alternatively over an Internet connection. Of course, the victim becomes suspicious as soon as the user data cannot be decrypted, but then it is too late as the evil maid already possesses the phone and the PIN.

We claim that classic evil maid attacks run on most Android devices given that the bootloader is unlocked. However, on smartphones where the bootloader is not unlocked, the networked variant must be taken into account. This variant also has the advantage that it requires only one physical access.

1.2.2 Armored: A Countermeasure against Cold Boot Attacks

With "Armored" [8], we developed a countermeasure against cold boot attacks on Android. CPU-bound implementations of AES are widely accepted as a protection mechanism to cold boot attacks [9, 10, 11, 12]. However, all solutions up to 2013 have been developed for PCs based on the x86 architecture. Armored is the first *CPU-bound encryption system* focusing on the ARM microarchitecture. More precisely, our contributions for Armored are:

- Armored runs the AES cipher [13] on ARM processors without involving RAM. The key, the key schedule and all intermediate values of AES are entirely stored in CPU registers. Due to the missing AES-NI instruction set [14], which is available on CPUs from Intel, it was unclear if such an implementation is possible on ARM smartphones. We solved this task by implementing AES with Gladman's method [15] in ARM assembly language and by exploiting ARM's multimedia register set NEON.

- CPU-bound encryption must be run in kernel mode to avoid side effects like context switching that move registers into RAM. Therefore, we provide Armored as a loadable kernel module (LKM) for Android. The Armored LKM must be inserted with root privileges and, as a consequence, our solution cannot be easily installed like a third party app for Android, but requires the underlying OS to be modified.

Finally, we provide a security and performance analysis of Armored. We prove that no critical state of AES ever enters RAM by observing a smartphone's main memory at runtime. On the downside, Armored runs twice as slow as Android's ordinary disk encryption. However, we regard this drawback acceptable for many practical use cases when comparing it to the gain in security.

1.3 Paper Outline

The remainder of this paper is structured as follows: In Sect. 2, we give background information about our device under test (the Samsung Galaxy Nexus), and the encryption feature in Android. Moreover we show related work regarding the security of Android's full disk encryption feature. In Sect. 3, we show two common attack scenarios, namely cold boot and evil maid attacks on full disk encryption, and how they can be applied to smartphones. In Sect. 4, we present countermeasures against cold boot attacks and give a detailed description of our implementation as well as an evaluation. Finally, in Sect. 5, we summarize our work and conclude with a discussion about limitations and future research directions.

2 Background and Related Work

We now give a brief history and technical description of Android's FDE feature (Sect. 2.1), specifically for Galaxy Nexus devices (Sect. 2.2), and present related work for the field of FDE security (Sect. 2.3).

2.1 Android's Full Disk Encryption Feature

Android is a Linux-based operating system from Google that can be used freely by any smartphone manufacturer. Android 1.0 was released in November 2007, and since version 4.0 (alias *ice cream sandwich*) which was published in October 2011, Google provides a disk encryption feature for the user partition. The encryption feature in Android is based on Linux' solution *dm-crypt* that is available in Linux kernels. Dm-crypt relies on the *device-mapper* infrastructure and the *Crypto API* of the Linux kernel. Although dm-crypt is suitable for *full disk encryption* (FDE), Android does not encrypt full disks but only user partitions. Dm-crypt supports different ciphers and modes of operation, most notably AES and CBC. Android 4 makes use of the cipher mode `aes-cbc-essiv:sha256` with 128-bit keys. The AES-128 *data encryption key* (DEK) is encrypted with an AES-128 *key encryption key* (KEK), which is in turn derived from the user PIN through the *password-based key derivation function 2* (PBKDF2) [16]. Using two different keys, namely the DEK and the KEK, renders reencryption in the case of PIN changes unnecessary.

Unlike iOS, which automatically activates disk encryption when a PIN is set, Android's encryption feature is disabled by default. Activating it manually takes up to an hour for the initial process and cannot be undone. Furthermore, it can only be activated if PIN-locks or passwords are in use. In Android, PINs consist of 4 to 16 numeric characters, and passwords consist of 4 to 16 alphanumeric characters with at least one letter. New screen locking mechanisms like pattern-locks and face recognition are less secure, and so Google forbids them in combination with disk encryption. Pattern-locks, for example, can be broken by *Smudge Attacks* [17], and face recognition can simply be tricked by showing a photo of the smartphone owner [18]. PINs are still the most frequent screen lock in use, but long PINs are inconvenient and most people use short PINs of only four digits. The reason is that a PIN must be entered for each separated interaction with the device like giving a call, writing a message, or taking a photo. However, short PINs are dangerous. In 2012, Cannon and Bradford [19] presented details about Android's encryption system and gave instructions on how to break it with brute force attacks.

2.2 Samsung Galaxy Nexus

We have chosen the Galaxy Nexus from Samsung for our tests because it is an official developer phone and was the first smartphone available with Android 4 (i.e., it was the first Android smartphone with a built-in encryption feature). Specifically, we own a GSM/HSPA model (GT-9250) codename *tuna/maguro*. The Galaxy Nexus comes with an OMAP4 chip from Texas Instruments (4460) which has a Cortex-A9 CPU implementing ARMv7.

The partition layout of an encrypted Galaxy Nexus is given in Table 1. Most of the thirteen partitions can be ignored for our purpose, except the partitions *mmcblk0p10* to *mmcblk0p13*, i.e., the system, cache, userdata, and metadata partition. The userdata partition contains the encrypted filesystem that we want to break and protect, respectively. The metadata partition is a crypto footer that holds necessary information about the encryption. The system partition holds the (unencrypted) Android operating system that we modify with EvilDroid and Armored. And the cache partition is the partition that we have chosen to save the logged PIN of EvilDroid.

To modify a system partition, the bootloader must be unlocked. Unlocking the bootloader can always be done with physical access. However, on Galaxy Nexus devices, the unlocking process deletes the

block device	partition name	description
/dev/block/mmcblk0p1	xloader	bootloader code
/dev/block/mmcblk0p2	sbl	bootloader code
/dev/block/mmcblk0p3	efs	static information like IMEI
/dev/block/mmcblk0p4	param	boot parameters
/dev/block/mmcblk0p5	misc	system settings like carrier ID
/dev/block/mmcblk0p6	dgs	unknown *(zero filled on all devices)*
/dev/block/mmcblk0p7	boot	boot code
/dev/block/mmcblk0p8	recovery	recovery image
/dev/block/mmcblk0p9	radio	radio firmware (GSM)
/dev/block/mmcblk0p10	**system**	Android operating system
/dev/block/mmcblk0p11	**cache**	cache (e.g., for user apps)
/dev/block/mmcblk0p12	**userdata**	user data (encrypted)
/dev/block/mmcblk0p13	**metadata**	crypto footer

Table 1: Partition layout of an encrypted Samsung Galaxy Nexus device.

userdata and cache partition. We verified that Google actually *wipes* the userdata and cache partition, meaning that these partitions get overwritten with zeros. However, it is generally a device-dependent property whether a bootloader is locked or unlocked, and whether partitions get wiped during unlocking or not. The first series of Galaxy Nexus devices did *not* wipe user partitions when unlocking the bootloader [20]. Other devices like the Samsung Galaxy SII are shipped with unlocked bootloaders out-of-the-box, such that unlocking is never necessary. Some people also unlock their bootloaders to "root" their devices, as required for Armored, even though this is not the standard configuration.

2.3 Related Work

On PCs, several attacks against software-based disk encryption are known, including *DMA attacks* [21] by Dornseif, Becher, and Klein (2005), *cold boot attacks* [6] by Halderman et al. (2008), and *evil maid attacks* [7] by Rutkowska (2009). DMA and cold boot attacks aim at the disk encryption key in RAM and so they require the target PC to be running (or at least to be in standby). Contrary to that, evil maid attacks aim at the user password and can be deployed against switched-off PCs.

2.3.1 Evil Maid Attacks

Traditional evil maid attacks require access to the target PC twice. Their name originates from the following scenario first described by Rutkowska (*"Evil Maid goes after TrueCrypt"* [7]): A hotel guest leaves an encrypted laptop back in the room and goes out for dinner. In the meantime, an "evil maid" can gain access to the room unsuspiciously and subvert the MBR (*master boot record*) of the laptop with keylogging. The MBR of an encrypted laptop must necessarily be left unencrypted for bootstrapping. Later, the hotel guest returns, accesses the manipulated machine and types in the password as usual, but this time the password gets logged. As soon as the guest leaves the hotel room again, the evil maid can read out the password and decrypt the entire machine.

Rutkowska implemented this attack practically against TrueCrypt, but earlier, in 2009, another bootkit already circumvented TrueCrypt, too, which became known as the *"Stoned Bootkit"* [22]. Even in 2013 TrueCrypt is still vulnerable against such bootkits and evil maid attacks. By contrast, BitLocker defeats

MBR manipulation attacks up to a certain degree by ensuring the integrity of the MBR by means of the TPM (*trusted platform module*). The decryption key of BitLocker can only be derived from the password if the machine's MBR has not been manipulated. Otherwise, the decryption key is unknown and the data on disk cannot be decrypted. However, TPMs are not available on Android smartphones, such that Android can be attacked similar to TrueCrypt.

In 2011, the possibility of evil maid attacks against smartphones was mentioned by Defreez for the first time [23]. Defreez focused on *WhisperCore*, which was a security oriented distribution of Android that introduced disk encryption to Nexus One and Nexus S devices before official Android releases did. However, results for WhisperCore have limited consequences in practice, because most people use official Android releases. Moreover, this work does not mention the possibility of "networked" evil maid attacks in the face of locked bootloaders as we do with EvilDroid.

2.3.2 Cold Boot Attacks

CPU-bound implementations of AES for the resistance of disk encryption against cold boot attacks are a well studied method in academia. The key as well as the key schedule, and the encryption process itself, can be confined to the CPU such that no encryption information is ever released to RAM. For x86 platforms, several such solutions exist that we briefly describe in the following.

In 2009, the idea of the first ever mentioned CPU-bound encryption system named *FrozenCache* [9] was illustrated by Pabel. FrozenCache was designed to hold keys in CPU caches, but it has never been implemented due to technical peculiarities dealing with CPU caches. CPU caches cannot be controlled well by the system level programmer as they are designed to act transparently. In 2010, a more practical solution to the cold boot problem became known as *AESSE* [24]. AESSE holds AES keys in SSE registers and is a working solution that is implemented as a Linux kernel patch. More solutions followed in the upcoming years with *TRESOR* [10], *LoopAmnesia* [11], and *TreVisor* [12]. All these solutions store necessary keys inside CPU registers, and – as long as no practical way to read out CPU registers is known – they are more secure than conventional disk encryption systems.

TRESOR (*TRESOR runs encryption securely outside RAM*) uses x86 debug registers to store AES keys such that they are not accessible from user space. Moreover, inside debug registers AES keys are secure against cold boot attacks. In detail, the four breakpoint registers $dr0$ to $dr3$ are used as cryptographic key storage. On 64-bit systems, this gives a total storage of $4 \cdot 64 = 256$ bits, enough to accommodate AES-256. On the downside, hardware breakpoints cannot be set by debuggers anymore. Other x86 registers, like SSE and general purpose registers, are utilized to execute the AES algorithm. These registers are used inside atomic sections only. Before leaving an atomic section, non-debug registers are reset to zero and consequently, they never enter RAM. Additionally, TRESOR makes use of Intel's *AES instruction set* [14] (AES-NI) in order to compute AES rounds efficiently on the CPU. This instruction set is available on Intel Core i5 and i7 processors, but not on current ARM processors, such that we had to solve additional challenges for Armored.

3 Attacks on Android's Full Disk Encryption Feature

In this section we cover two possible attack scenarios on Android's full disk encryption feature. The first attack scenario are cold boot attacks; we give a brief summary of the recovery tool FROST [25], which proves that these attacks are feasible against smartphones, too. The second scenario are evil maid attacks; we show that they are possible on Android smartphones for the first time.

3.1 FROST: Cold Boot Attacks

As shown by FROST [25], it is possible for an unauthorized party with physical access to an encrypted Android phone to recover its data using cold boot attacks. *Cold booting* a device technically means to briefly cycle power off and on without allowing the OS to shut down properly. But there is also a second meaning of the term "cold". RAM chips of PCs and smartphones exhibit a behavior called the remanence effect. The remanence effect says that RAM content fades over time rather than disappearing all at once. An interesting fact is that contents fade more slowly at lower temperatures. The colder RAM chips are, the longer their memory contents persist. Hence, cold boot attacks are more practical when the target device is *cold*.

To exploit this behavior, Müller and Spreitzenbarth developed the recovery tool FROST. If an adversary gains access to a phone's main memory before it fades completely, he or she is able to reconstruct valuable information from RAM with FROST. This information includes personal messages, calendar entries, photos, and the disk encryption key. FROST requires an adversary to cold boot the target device by replugging its battery quickly, because smartphones usually have no reset button. The battery must be replugged fast since the phone must be without power for less than a second. Otherwise, the bits in RAM begin to decay and a significant part of data gets lost. To increase the remanence interval, and correspondingly to increase the success rate of their attack, the authors suggest putting the target phone into a $-15°C$ freezer for 60 minutes before replugging the battery. The operating temperature of a phone, which is usually around $30°C$, then decreases to less than $10°C$, significantly raising the chance for key recovery. Below $10°C$, only 5% of RAM bits are decayed after one second, whereas higher temperatures yield less reliable results.

After replugging the battery of the phone, FROST must be installed into the recovery partition via USB and then booted up. Once FROST is running, it can be used to acquire full dumps of a smartphone's RAM, to recover the disk encryption key, and to decrypt the user partition. Even though the authors state that FROST can break disk encryption only if the bootloader of a phone is unlocked and open for manipulations, which is not the case for all devices in practice, their result proves that cold boot attacks against ARM are a real threat. Basically, their approach is not limited to Android but affects, for example, the *Ubuntu Touch* and other ARM OSs as well. As a consequence, we believe it is important to develop a cold boot resistant encryption solution for the ARM microarchitecture.

3.2 EvilDroid: Evil Maid Attacks

We have chosen the smartphone OS Android as target for our evil maid attacks because it is most amenable for an in-depth security analysis since it is open-source and Linux-based. With open-source Android, modifying the kernel sources – as required by our implementation – becomes possible.

The weak point of Android's encryption feature is its unencrypted system partition. In Android, only the user partition is encrypted but the system partition remains open for manipulations. Contrary to that, BitLocker and TrueCrypt pursue a two-staged boot sequence where the system partition is decrypted by means of a pre-boot environment. Therefore, at least the integrity of the system partition is given by BitLocker and TrueCrypt, although the integrity of the pre-boot environment is not guaranteed by these solutions. In Android, however, there is only one encryption stage, such that even the integrity of the system partition is not guaranteed. As the system partition itself asks users for their passwords, Android is widely open to system manipulations.

Basically, we replace the entire Android system with our own variant of Android that additionally performs keystroke logging. Installing our own variant of Android can be done with physical access to a Galaxy Nexus device, i.e., no user privileges for the device are required. However, if the bootloader is locked, we must unlock it before the installation. In Sect. 3.2.1, we assume the bootloader is already unlocked before accessing the phone. This enables us to deploy *classic evil maid attacks*. In Sect. 3.2.2, we discuss the case of locked bootloaders, forcing us to use *networked evil maid attacks*.

3.2.1 Classic Evil Maid Attacks

While third party apps that extend the functionality of Android are primarily written in Java, Android's encryption feature and PIN prompt reside in system space which is written in C. Roughly speaking, we patch the original boot code to additionally log the user PIN. There is no need to emulate Android's password prompt, because we have the source code of Android that we can simply manipulate.

In detail, we patch the file `/system/vold/cryptfs.c` of the Android source code. This file includes the code for displaying visual PIN prompts. After a PIN is entered, Android derives a *key encryption key* (KEK), decrypts the *data encryption key* (DEK), and then checks whether the user partition can be decrypted with the DEK. If so, the correct PIN has been entered, and at this point we log it. To be exact, we write the logged PIN to `/cache/.evilpw` on the unencrypted cache partition. This file can afterwards easily be read out via ADB (*android debugging bridge*).

After we apply these changes to the Android source, we recompile the Android system image (called `evildroid.img`). This image outwardly appears exactly like the original Android image from Galaxy Nexus devices. The difference, however, is that the PIN gets logged in background as soon as it is entered by the legitimate user. We install this modified version of Android to a target device by connecting it to the PC via USB and then running the command `fastboot flash system evildroid.img`. The PIN can then be read out by connecting the phone again and running the command `adb pull /cache/.evilpw`. If ADB is not activated, it can be activated by installing a custom recovery ROM (e.g., ClockworkMod). All steps can be done with physical access only and do not require user privileges.

To sum up, in the classic evil maid variant, two physical accesses are required per smartphone. On the first physical access, the system partition is manipulated in a way that it logs the user PIN. The user PIN is then stored on an unencrypted part of the disk. On the second physical access, this PIN is read out and the phone can be accessed afterwards, and, for example, a one-to-one copy of the user partition can be acquired. The classic evil maid variant has the advantage that ordinary users have no chance to recognize the fraud. However, this variant has the disadvantage that the bootloader must be unlocked or that it must be possible to unlock the bootloader without wiping the phone. Otherwise, it is impossible to install a manipulated Android OS to the system partition.

3.2.2 Networked Evil Maid Attacks

If the bootloader of a Galaxy Nexus is locked, we run `fastboot oem unlock` before we can install the EvilDroid image. This can always be done with physical access only, but it requires us to confirm the following warning on the phone: "To prevent unauthorized access to your personal data, unlocking the bootloader will also delete all personal data from your phone". That is, if we want to unlock the bootloader, we have to accept that the user data gets deleted. As stated above, we verified that the user partition is actually *wiped*, i.e., that it gets overwritten with zeros. We suppose that it is not possible to modify this unlocking procedure, because it is part of the read-only firmware memory.

Obviously, wiping the user partition renders attacks against the encryption useless. Hence, we have to consider other ways to trick users into disclosing their passwords. To overcome the need for an unlocked bootloader, we *replace* the target device with an identical model. Only one physical access is required then, because instead of collecting the PIN with a second physical access, we receive the PIN via SMS, or over an Internet connection. Indeed, users can now immediately identify the fraud because after entering their PINs, the user environment cannot be displayed anymore. However, when the user identifies the fraud it is already too late because the evil maid possesses the encrypted phone as well as the PIN. The underlying security problem is that the authenticity of the phone cannot be proven towards the user at the time of entering the disk encryption password. Custom wallpapers, for example, are displayed in the screen lock prompt, but not in the disk encryption prompt. During boot, the user wallpaper is encrypted and consequently, it cannot be displayed by the system at this time. (Note that a custom wallpaper would not be a valid authentication method anyway, as it can easily be forged. For a brief overview about countermeasures based on tamper-proof "mutual authentication" of systems and users, see Sect. 5.1.)

From a technical point of view, we modified the boot prompt of Android by patching the official C sources, similar to the classic variant. This time, we can write the password to `/data/.evilpw` instead of the cache partition. In addition to that, we implemented a user mode app written in Java that reads the logged password from the disk and sends it to a predefined mobile number via SMS, or to an IP address via Internet. This app must be added to the auto start programs of the manipulated phone. Having the opportunity to send PINs over nearby WiFi connections, rather than over SMS, also has the advantage that devices without SIM are equally affected.

By design, evil maid attacks require some kind of *social engineering* and thus, they cannot be classified as pure technical approaches. That is particularly the case for the networked variant because the legitimate user has to be fooled to regard a foreign phone for his or her own phone. For example, if a phone is partially broken (like a scratch in the glass), the attack becomes practically harder because the owner has a second, non-digital way to identity the phone as his or her own. However, since the Android PIN prompt at boot time is not personalized, meaning that personal background images are only shown after boot, a digital way to identify a phone is missing, such that the attack is relevant for many practical scenarios.

4 Armored: A Countermeasure against Cold Boot Attacks

The fact that cryptographic keys in main memory are unsafe has been known for a long time. Nevertheless, all vendors of software-based encryption solutions continue to store cryptographic keys inside RAM, including Google's Android OS. Most likely, vendors believe that running encryption and key management without RAM is impossible, or at least very costly, or that it requires dedicated hardware.

One possibility to overcome the threat of cold boot attacks while keeping keys in main memory is to detain adversaries from accessing RAM content. This can be done, for example, by immutable boot sequences and soldered RAM chips as it is the case for Apple's iPhone series and many Windows phones. However, as shown by FROST, manufacturers of Android smartphones began to provide open bootloaders that can be unlocked with physical access. In such systems, the disk encryption algorithm must essentially be executed *outside RAM* in order to provide resistance against cold boot attacks, as we do in Armored.

4.1 Design and Implementation

We now give details about the first CPU-bound encryption system for ARM. We implemented Armored mostly in ARM assembly language, because high level languages like C make use of the heap and stack. We are not allowed to use these memory regions, because in Armored we follow a strict security policy: no state or intermediate state of AES, including all runtime variables, is ever allowed to go to RAM. This security policy obviates future crypt analyses exploiting intermediate values of AES in RAM.

Many of the ideas we applied in Armored are based on what we learned from TRESOR [10]. However, not much from the actual TRESOR code could be applied in Armored, because TRESOR is an architecture dependent implementation for 64-bit x86 PCs. Contrary to that, Android-based smartphones are 32-bit ARM devices. The difficulty in particular was that ARM processors have no AES-NI instruction set. Therefore, we had to implement the AES algorithm for ARM from scratch without hardware support of the processor. Note that various AES implementations for ARM exist, for example by Bernstein and

Schwabe [26], but that we cannot directly benefit from these implementations because they do not avoid the use of RAM.

Nevertheless, we have chosen TRESOR as the basis for our implementation (and learned a lot from it regarding its integration into the Linux kernel) because TRESOR is implemented as a dm-crypt module for Linux. We implemented Armored as a dm-crypt module, too, because Android's encryption feature is based on dm-crypt. Other solutions to the cold boot problem, such as LoopAmnesia [11] and TreVisor [12], are not derived from dm-crypt, and consequently, they are not suited as a basis for our implementation.

4.1.1 Key Storage Registers

The first challenge we had to solve was to find a register set available on ARM CPUs that is qualified as AES key storage. Since key storage registers for Armored must permanently be occupied and cannot be used for their intended purpose, we had to choose them carefully. Unprivileged registers were automatically disqualified because they are essential for third party apps and, even worse, their content is periodically written into RAM as part of context switching. Instead, we came up with a mixed set of ARM-specific breakpoint and watchpoint registers, because those are (1) only accessible from kernel mode, and (2) seldom used by end-users. But unlike the debug registers in 64-bit x86 CPUs, they are too small to hold AES-256 keys. (Note that debug registers can be written into RAM due to context switching as well, but we specifically prohibit that by patching respective kernel routines.)

On ARM, the least significant two bits of each 32-bit break- and watchpoint register are necessarily zero due to the memory alignment in ARM. Since instructions are consistently 32-bit wide, they are always located at 4-byte aligned addresses. Hence, the least significant two bits are omitted for setting break- and watchpoints because they must be zero anyway. As a consequence, these bits are not available as key storage. For the sake of convenience, we divide the key-sequence into 16-bit chunks; more detailed, we use four breakpoint and four watchpoint registers, giving us a total of $8 \cdot 16 = 128$ bits as key storage. This is enough to accommodate AES-128, but not enough to accommodate AES-256. However, since Android's encryption feature is based on AES-128, this does not pose a problem in practice.

In subsequent releases of Armored we could store more than 16 bits per register, and if we find additional break- and watchpoint registers, we could accommodate AES-256. ARM is more a construction kit for CPUs than a definite regulation for registers and instructions. The number of break- and watchpoint registers depends on the specific platform; four seems to be the "minimum" that is commonly available. On our development platform (a PandaBoard) we have six break- and four watchpoint registers.

4.1.2 NEON Multimedia Registers

Besides debug registers, Armored is based on the multimedia register set NEON, which is available on ARM CPUs like the Cortex-A9 series. NEON is a SIMD (*single instruction multiple data*) extension providing parallel 64-bit and 128-bit operations on ARM. NEON features its own instruction set and has an independent execution hardware, but most notably, it has a separate register set. This register set encompasses sixteen 128-bit registers, i.e., 2 kilobits in total, which are also addressable as 64-bit registers. Roughly speaking, we use these registers as a surrogate stack or heap for our AES implementation, because we do not want to use RAM.

Transferring data between general purpose registers and NEON registers can be done on byte level, just like storing data in memory. Moreover we can directly perform SIMD instructions on NEON registers that could not be performed on memory locations. For accessing NEON registers safely in our algorithm, we run encryption and decryption steps inside *atomic sections*. Inside these sections, our code cannot be interrupted, neither through preemption from scheduling nor through hardware interrupts. When we take care to reset NEON registers before leaving atomic sections, we do not leak sensitive information into RAM but can use these registers in a secure manner during our AES algorithm.

Note that *non-maskable interrupts* (NMIs) cannot be deferred by software-based atomic sections as used in Armored. However, NMIs are mostly caused by hardware failures and hence, often lead to a kernel panic. Up to now, we ignore the threat of NMIs because we find it unlikely that (1) an attacker can induce a hardware failure that (2) leads to an NMI at the precise moment when Armored is active and (3) can perform a cold boot attack in this short time frame. However, if NMIs turn out to pose a problem, a possible countermeasure would be to patch all NMI handlers of the OS in a way that the CPU context is not saved. Instead, the machine could be halted after an NMI event occurs.

4.1.3 Gladman's AES Method

Unlike TRESOR, which relies on Intel's AES-NI instruction set, we had to implement AES manually. On x86 CPUs, the CPU instruction *aesenc* performs an entire AES round, and, broadly speaking, TRESOR just calls aesenc 10 times to encrypt one AES-128 block. Unfortunately, it is not as easy on ARM, but we have to make use of the AES method invented by Gladman [15]. Gladman's AES method is based on table lookups, and it is both efficient and qualified for the use with only a few registers. An additional burden with ARM, however, is that the base address of a lookup table cannot be loaded into a register directly as a 32-bit immediate value. The problem is that ARM instructions, unlike x86 instructions, must be exactly 32-bit wide, including the opcode and three operands. Consequently, 32-bit immediate values are not possible. To overcome this issue, we have to generate a pool of constants near the encryption routine and used a PC-relative, indirect addressing mode with shorter immediate values.

Another implementation detail is the need to recompute AES key schedules per atomic section. In common AES implementations, the key schedule is computed once and then stored inside RAM for performance reasons. But in Armored, the break- and watchpoint registers are occupied with the AES key and there is no space left to store round keys. Therefore, we have to recompute round keys for each atomic section. This is the main reason for the performance drawback of Armored as compared to generic AES (see Sect. 4.2.1). TRESOR faces a similar problem, but again, TRESOR benefits from Intel's AES instruction set: calling *aeskeygenassist* suffices to generate the next round key, such that TRESOR's performance does not suffer much.

For the sake of simplicity, we do not list our ARM code here. The overall kernel patch has 1700 lines of code from which about 500 lines are assembly code for the AES method. The remaining code is primarily written in C and is required to integrate our algorithm into the kernel and to handle modes of operations like CBC. As stated above, we implemented the AES-128 variant of Gladman's algorithm, because this is the relevant variant for Android. After developing Armored on a PandaBoard, we tested it successfully on real smartphones with OMAP4 chips from Texas Instruments. This chip is built into devices like the Samsung Galaxy Nexus.

4.2 Evaluation

We evaluated Armored regarding its performance (Sect. 4.2.1), its usability and correctness (Sect. 4.2.2), and its security (Sect. 4.2.3).

4.2.1 Performance

CPU-bound encryption schemes are known to suffer from necessary on-the-fly computations of the AES key schedule. On x86 systems, a performance drawback of factor 2.04 is stated for LoopAmnesia [11], and with AESSE [24] a performance drawback between factor 2.27 and 6.93 is given. Contrary to that, TRESOR [10] and TreVisor [12] have a performance drawback of "only" up to 50% in comparison to generic AES, because they utilize Intel's AES instruction set. With Armored, however, we faced additional problems arising from the CPU architecture, and our implementation had a consistent performance drawback of factor 4 to 5 at the beginning.

We were able to decrease the performance drawback of Armored down to factor 2.3 with the following innovation: All CPU-bound encryption systems to date begin a new atomic section per AES input block. That means, AES key schedules must be recomputed for every 128 bits. This solution is most straightforward from an implementation point of view, but to increase performance we propose larger atomic sections. In Armored, we expand the scope of an atomic section to 16 AES input blocks, i.e., we recompute round keys of 2 kilobits each. As a consequence, only one-sixteenth of the key schedule computations are required in comparison to earlier implementations. This improvement raised the throughput of Armored to 6.76 MB/s in relation to 15.55 MB/s with generic AES, i.e., to factor 2.3. (We measured the absolute values on a PandaBoard development environment by reading 400 MB random data blocks from an encrypted RAM disks.)

The interesting question with this improvement was: How many blocks can be encrypted within one atomic section until we get interference with the interactivity of multitasking systems? To answer this question, we measured the average time that is required to encrypt single AES blocks, and we have observed that these times are in the range of 1 to 2 microseconds. Contrary to that, Linux scheduling slices are about 50 milliseconds and thus, we consider atomic sections of up to 1024 input blocks as safe. We have chosen "only" 16 blocks in our implementation, because the performance gain from larger atomic sections is minimal. One-sixteenth of the overhead is already very small. We have tested that our processor bound implementation reaches at most 7 MB/s without round key recomputations, such that 6.67 MB/s can be considered near-optimal.

We enhanced the important mode of operation CBC [27] (*cipher block chaining*) to process 16 blocks at once. Multiple CBC blocks are not processed in parallel, but they are processed inside the same atomic section such that the key schedule can be computed once. Technically, we export `cbc(armored)` to the Linux crypto API, and assign a higher priority to it than to the generic versions of AES and Armored. That means, whenever CBC is used and more than one block is waiting – which is usually the case in full disk encryption – our optimized variant is automatically executed by the Linux kernel.

4.2.2 Usability and Correctness

At the time of this writing, we copy the secret AES key into the debug registers of a smartphone via ADB (Android debug bridge). To this end, we must connect the phone with USB to a computer, login as root, and write a sequence of keybits via Linux' sysfs interface into the debug registers. Afterwards, we run a cleanup procedure to remove all key residues from RAM. We provide a small userland utility that writes the key via sysfs into the kernel and runs the cleanup procedure automatically. According to our security tests (see Sect. 4.2.3), no key residues are left behind.

Admittedly, this process is impractical for end-users, and that is why we consider Armored as a proof-of-concept implementation. To make the implementation more practical, Android's graphical PIN or password prompt must be patched in a way that the encryption key is directly written into debug registers. Although we did not implement this feature, we believe it is possible, and we believe it would make processor bound encryption available to a large number of end-users.

To prove the cryptographic correctness of Armored, we used the official test vectors for AES that are listed in FIPS-197 [13]. Armored is integrated into the Linux crypto API in a way that the kernel *test manager* verifies the correctness of Armored based on these vectors each time the module is loaded. Moreover, we encrypted user partitions with Armored, decrypted them with generic AES and vice versa. Along with structured data like text files, we created large random files on these partitions. We compared the unencrypted versions of the files and found them to be equal. This is a strong indication for the correctness of our implementation, because it does not only prove the correctness in terms of predefined test vectors, but also regarding a great amount of random data.

4.2.3 Security

To evaluate Armored's resistance against cold boot attacks, we reproduced cold boot attacks against Samsung Galaxy Nexus devices with FROST. As we expected, we could not recover the key, the key schedule, or any intermediate state of AES. However, key recovery by FROST had to fail because it bases on the AES key schedule that we entirely discard in Armored. That is, even though Armored would accidentally leak the secret key into RAM (e.g., because of context switching) we would not be able to recover the key with FROST. Hence, we searched for known patterns of the key directly in RAM, because (unlike real attackers) we know the secret key in advance. Again, we acquired memory dumps by means of FROST, and additionally we acquired memory dumps from running devices with the forensic module LiME [28]. With LiME, we were able to observe a smartphone's RAM at runtime without the need to actually reboot the phone. Thereby, we had more reliable results because cold boot attacks are error-prone and may falsify parts of the key. As we expected, also when observing RAM at runtime we could not find significant matches of the key or parts of it. We observed the RAM multiple times and at different system states during our tests.

CPU-bound encryption systems defeat cold boot attacks, but they are vulnerable to attackers who have write access to the system space. An attacker with root privileges, for example, can easily load a kernel module that moves the key from CPU registers into main memory. However, given that our environment guarantees kernel integrity, CPU-bound encryption systems are more secure. If there is no way to write into system space, there is no way (that we know of) to recover key bits from CPU registers.

Blass and Robertson refer to this property as the *kernel integrity property* [29]. This property says that attackers are disallowed to execute code in the context of the kernel. Unfortunately, such an assumption is hard to generalize for x86 PCs. In practice, malware can often gain root privileges, and DMA attacks can write into system space through physical access. Indeed, CPU-bound encryption schemes do not depend on the integrity of the kernel, because they are primarily invented to defeat cold boot attacks, but this property is preferable. For Android smartphones, however, we believe that the desired kernel integrity property can be guaranteed up to a certain degree for the following reasons:

- System privileges: In Android, the root account is disabled by default and can be re-enabled only by installing a custom Android ROM. On regular Android phones, there is no possibility to load kernel modules for both designated users and potential attackers. Besides LKMs, Linux users can write into system space via */dev/kmem*. This device is disabled on Android as well.

- Direct memory access (DMA): Another way to write into system space are DMA attacks. Solutions like TRESOR and Armored cannot protect against DMA attacks on running machines, as proven in 2012 [29]: DMA ports like FireWire [21] and Thunderbolt [30] allow compromising the system space and violating kernel integrity. However, DMA ports are commonly not available on smartphones and USB ports are not DMA capable.

To sum up, CPU-bound encryption generally protects against cold boot attacks but not against attacks manipulating the system space. Only if we additionally take integrity of the kernel for granted, which is difficult in general, CPU-bound encryption can defeat more attacks. We believe that Android-based smartphones without enabled JTAG interface come close to the *kernel integrity property*. Indeed, our argumentation is based on the fact that root accounts are not available on Android, but it ignores the threat of privilege escalations due to kernel bugs.

5 Discussion and Conclusions

To conclude, we discuss some final remarks regarding EvilDroid (Sect. 5.1) and Armored (Sect. 5.2), and finally give a short summary (Sect. 5.3).

5.1 EvilDroid Discussion

Classic evil maid attacks are mostly defeated by simply keeping the bootloader locked. As it is possible on most devices to re-lock a bootloader after it has been unlocked, we recommend users who want to "root" their device to re-lock it. However, the simplicity and effectiveness of networked evil maid attacks highlight the importance of *physical security* for mobile devices. Evil maid attacks are always possible if systems and users cannot properly authenticate each other mutually. Indeed, the DEK in Android can only be derived after a user has been authenticated but looking at the authentication the other way round, users have no chance to verify the trustworthiness of the boot process of a smartphone. Mechanisms as enforced by BitLocker, for example, are not yet possible for smartphones, because they have no TPMs at the time of this writing.

Unfortunately, also if TPMs would be build into smartphones, the networked evil maid attack could still be deployed. The problem is that, despite of the use of a TPM, the trustworthiness of the PIN prompt cannot be decided by the user. Consequently, more complex countermeasures have to be taken into consideration. Advanced countermeasures known from PCs are Anti Evil Maid by Rutkowska [31] and STARK by Müller et al. [32]. These schemes mention *mutual authentication* in terms of disk encryption for the first time. They build upon the TPM, too, but go one step further than ordinary solutions: if the boot process behaves with integrity, a secret (user-defined) message is unsealed by the TPM and then displayed to the user. If this message cannot be displayed to the user, the user is strongly advised not to enter a password because the boot process is likely to be compromised. However, on PCs the sealed messages are usually stored on external USB drives (*trust bootstrapping*), what is not an option for smartphones.

Altogether, we have shown that it is possible to perform evil maid attacks against Android devices and countermeasures are apparently difficult. Hence, our work must be considered as a warning to both users and manufacturers to lock bootloaders and to disable options to unlock them without wiping the user partition. The probably most important fact we are able to learn from our analysis is that a trusted boot process [33] will be necessary for mobile devices as it is already the case for desktop PCs and laptops. Only with the support of new hardware modules it is possible to defeat evil maid attacks such that trusted computing needs to be deployed on mobile devices, too.

5.2 Armored Discussion

CPU-bound encryption like Armored can "only" secure the disk encryption key. Other RAM contents, such as contact lists, calendar entries, personal messages, and browser caches, remain unencrypted in RAM and are thus still accessible for an adversary. So the most notable limitation of Armored is that it cannot protect information other than disk encryption keys (but other CPU-bound encryption systems face the same limitation).

Another notable limitation of Armored is its performance. The advantage of Armored, that it runs on all ARM CPUs with a NEON instruction set, comes at the cost of encryption speed. However, this limitation might be overcome by supporting dedicated crypto instructions and hardware accelerators from advancements in available ARM hardware. For example, since ARMv8 (e.g., Cortex-A15) specialized AES and SHA instructions are supported directly by the main CPU. Most likely, Armored can benefit from these instructions, leading to a higher encryption throughput.

To defeat cold boot attacks for ordinary end users, systems like Armored must be integrated into official releases of Android. Basically, Armored exports a cold boot resistant interface of AES-128 to the Linux dm-crypt API, making it available for Android encryption. In practice, however, Armored must be considered as work in progress, because it is not well integrated into Android's boot process yet. The problem during bootstrapping is: How do we get the initial encryption key from end-users into debug registers? This must be handled securely by Android's PIN prompt.

5.3 Summary

Android devices with an unlocked bootloader are vulnerable to classic evil maid attacks because the system partition is unencrypted. Even worse, all Android devices are vulnerable to networked evil maid attacks, because the pre-boot environment displays a uniform PIN prompt. Users have no chance to digitally identify a fraud before entering a PIN because prompts outwardly appear identical on similar Android phones. The vulnerability we exploit actually arises from the fact that PIN prompts in Android are not tamper-proof such that users can easily be tricked into typing their passwords into bogus prompts.

With Armored, we have presented the first CPU-bound encryption system for ARM. In comparison to its PC-based counterpart TRESOR, Armored inherits some advantages from Android. Firstly, a point of criticism against TRESOR is that it supports only single encryption keys, but in Android there is just one encrypted partition anyway. Secondly, another problem of TRESOR is handling the ACPI mode S3 (suspend-to-RAM). Since CPUs are switched off during S3, the encryption key is lost and must be re-entered upon wakeup. Android devices, however, have no sleep mode that switches off the CPU.

To sum up, we have shown an in-depth analysis of the security of Android's full disk encryption feature by demonstrating both practical attacks like EvilDroid and safeguards like Armored. While we could successfully defeat cold boot attacks with the technology of CPU-bound encryption, defeating evil maid attacks is a more difficult task due to the lack of trusted platform modules in smartphones.

References

[1] Ponemon Institute LLC, "Smartphone Security: Survey of U.S. consumers," in *Ponemon Institute Research Report*. sponsored by AVG Technologies, January 2011, http://aa-download.avg.com/filedir/other/Smartphone.pdf.

[2] S. Skorobogatov, "Data Remanence in Flash Memory Devices," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES'05), Edinburgh, Scotland, LNCS*, vol. 3659. Springer-Verlag, August-September 2005, pp. 339–353.

[3] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. Burleson, and K. Fu, "TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices withou Clocks," in *Proc. of the 21th USENIX Security Symposium, (USENIX Security'12), Bellevue, WA, USA*. USENIX Association, August 2012.

[4] R. Anderson and M. Kuhn, "Tamper Resistance: A Cautionary Note," in *Proc. of the 2nd USENIX Workshop on Electronic Commerce (WOEC'96), Oakland, California, USA*. USENIX Association, November 1996, pp. 1–11.

[5] P. Gutmann, "Data Remanence in Semiconductor Devices," in *Proc. of the 10th USENIX Security Symposium (USENIX Security'01), Washington, DC, USA*. USENIX Association, August 2001.

[6] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold Boot Attacks on Encryptions Keys," in *Proc. of the 17th USENIX Security Symposium (USENIX Security'08), San Jose, California, USA*. USENIX Association, August 2008, pp. 45–60.

[7] Joanna Rutkowska, "Evil Maid goes after TrueCrypt," http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html, October 2009, The Invisible Things Lab.

[8] Johannes Götzfried and Tilo Müller, "ARMORED: CPU-bound Encryption for Android-driven ARM Devices," in *Proc. of the 8th IEEE International Conference on Availability, Reliability and Security (ARES'13), Regensburg, Germany*. IEEE, September 2013, pp. 161–168.

[9] J. Pabel, "FrozenCache – Mitigating Cold-Boot Attacks for Full-Disk-Encryption Software," in *27th Chaos Communication Congress (27C3), Berlin, Germany*, December 2009.

[10] T. Müller, F. Freiling, and A. Dewald, "TRESOR Runs Encryption Securely Outside RAM," in *Proc. of the 20th USENIX Security Symposium, (USENIX Security'11), San Francisco, California, USA*. USENIX Association, August 2011, pp. 17–17.

[11] P. Simmons, "Security Through Amnesia: A Software-based Solution to the Cold Boot Attack on Disk Encryption," in *Proc. of the 27th Annual Computer Security Applications Conference (ACSAC'11), Orlando, Florida, USA*. ACM, December 2011, pp. 73–82.

[12] T. Müller, B. Taubmann, and F. Freiling, "TreVisor: OS-Independent Software-Based Full Disk Encryption Secure Against Main Memory Attacks," in *Proc. of the 10th International Conference on Applied Cryptography and Network Security (ACNS'12), Singapore, LNCS*, vol. 7341. Springer-Verlag, June 2012, pp. 66–83.

[13] FIPS, "Advanced Encryption Standard (AES)," National Institute for Standards and Technology, Federal Information Processing Standards Publication 197, November 2001.

[14] S. Gueron, "Intel's New AES Instructions for Enhanced Performance and Security," in *Proc. of the 16th International Workshop on Fast Software Encryption (FSE'09), Leuven, Belgium, LNCS*, vol. 5665. Springer-Verlag, February 2009, pp. 51–66.

[15] B. Gladman, "A Specification for Rijndael, the AES Algorithm," www.gladman.me.uk, August 2007.

[16] M. Turan, E. Barker, W. Burr, and L. Chen, "Special Publication 800-132: Recommendation for Password-Based Key Derivation," NIST, Computer Security Division, Information Technology Laboratory, Tech. Rep., December 2010.

[17] A. J. Aviv, K. Gibson, E. Mossop, M. Blaze, and J. M. Smith, "Smudge Attacks on Smartphone Touch Screens," in *Proc. of the 4th USENIX Workshop on Offensive Technologies (WOOT'10), Washington, DC, USA*. USENIX Association, August 2010.

[18] M. Kumar, "Android facial recognition based unlocking can be fooled with photo," http://thehackernews.com/, November 2011, The Hacker News.

[19] T. Cannon and S. Bradford, "Into the Droid: Gaining Access to Android User Data," in *DefCon Hacking Conference (DefCon'12), Las Vegas, Nevada, USA*, July 2012.

[20] xdadevelopers, "Google Play Nexus not wiping after Bootloader Unlock," http://forum.xda-developers.com, April 2012, Thread 1650830.

[21] M. Becher, M. Dornseif, and C. N. Klein, "FireWire - All your memory are belong to us," in *Proc. of the Annual CanSecWest Applied Security Conference, Vancouver, British Columbia, Canada*, May 2005.

[22] Peter Kleissner, "Stoned Bootkit," in *Black Hat (BH'09), Las Vegas, Nevada, USA*, July 2009.

[23] D. Defreez, "Android Privacy Through Encryption," Master's thesis, University of Ashland, Department of Computer Science, May 2012.

[24] T. Müller, A. Dewald, and F. Freiling, "AESSE: A Cold-Boot Resistant Implementation of AES," in *Proc. of the 3rd European Workshop on System Security (EUROSEC'10), Paris, France*. ACM, April 2010, pp. 42–47.

[25] T. Müller and M. Spreitzenbarth, "FROST: Forensic Recovery Of Scrambled Telephones," in *Proc. of the 11th International Conference on Applied Cryptography and Network Security (ACNS'13), Banff, Alberta, Canada, LNCS*, vol. 7954. Springer-Verlag, June 2013, pp. 373–388.

[26] D. J. Bernstein and P. Schwabe, "NEON crypto," http://cr.yp.to/, March 2012, Department of Computer Science, University of Illinois at Chicago, USA.

[27] *Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode*, Addendum to special publication 800-38a ed., National Institute of Standards and Technology, 2010.

[28] Joe Sylve, "LiME - Linux Memory Extractor," in *ShmooCon'12, Washingtion, DC, USA*, January 2012.

[29] E.-O. Blass and W. Robertson, "TRESOR-HUNT: Attacking CPU-Bound Encryption," in *Proc. of the 28th Annual Computer Security Applications Conference (ACSAC'12), Orlando, Florida, USA*. ACM, December

2012, pp. 71–78.

[30] Break & Enter: Improving security by breaking it, "Adventures with Daisy in Thunderbolt-DMA-Land: Hacking Macs through the Thunderbolt interface," February 2012.

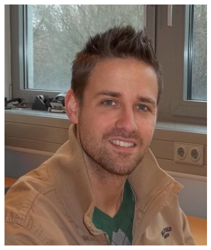
[31] Joanna Rutkowska, "Anti Evil Maid," http://theinvisiblethings.blogspot.de/2011/09/anti-evil-maid.html, September 2011, The Invisible Things Lab.

[32] T. Müller, H. Spath, R. Mäckl, and F. Freiling, "STARK Tamperproof Authentication to Resist Keylogging," in *Proc. of Financial Cryptography and Data Security (FC'13), Okinawa, Japan*. International Financial Cryptography Association (IFCA), April 2013.

[33] B. Parno, "Bootstrapping Trust in a Trusted Platform," in *Proc. of the 3rd USENIX Workshop on Hot Topics in Security (HotSec'08), San Jose, California, USA*. USENIX Association, July 2008.

Author Biography

Johannes Götzfried studies Computer Science at the University of Erlangen-Nuremberg (Germany). He received his B.Sc. in Computer Science from the University of Erlangen-Nuremberg with honors in 2012. Johannes has worked at the Fraunhofer Institute of Integrated Circuits in the field of ARM programming and currently works as a scientific coworker at the chair for IT Security Infrastructures headed by Prof. Dr. Freiling. He has published some articles at conference level in the field of mobile and hard disk security. His research interests include physical security, system security and mobile security.



Tilo Müller studied Computer Science at the University of Aachen (Germany) and the University of Helsinki (Finland). He received his Diploma from the University of Aachen with honors in 2010. Since then he is employed as a research assistant at the chair for IT Security Infrastructures at the University of Erlangen-Nuremberg. Tilo Müller received his doctoral degree in 2013 and now holds a postdoc position in Erlangen. He graduated about the practical security of full disk encryption against physical access attacks. Above that, his research interests include system security and mobile security.