# ARMORED
## CPU-bound Encryption for Android-driven ARM Devices

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

***Abstract*—As recently shown by attacks against Android-driven smartphones, ARM devices are vulnerable to *cold boot attacks*. At the end of 2012, the data recovery tool FROST was released which exploits the *remanence effect* of RAM to recover user data from a smartphone, at worst its disk encryption key. Disk encryption is supported in Android since version 4.0 and is today available on many smartphones. With ARMORED, we demonstrate that Android's disk encryption feature can be improved to withstand cold boot attacks by performing AES entirely without RAM. ARMORED stores necessary keys and intermediate values of AES inside registers of the ARM microprocessor architecture without involving main memory. As a consequence, cold boot attacks on encryption keys in RAM appear to be futile. We developed our implementation on a PandaBoard and tested it successfully on real phones. We also present a security and a performance analysis for ARMORED.**

***Keywords*-CPU-bound encryption, Cold boot, AES, ARM, Android**

## I. INTRODUCTION

Every few years, researchers warn against something we often forget – the *remanence effect* of main memory [1], [2]. The remanence effect says that RAM is less volatile than most people expect. 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 [3] proposed the first theoretic attack exploiting the remanence effect. Later in 2001, Gutmann [4] extended the basic idea by Anderson and Kuhn, and provided a more detailed insight into the remanence effect. As a consequence of his insights, Gutmann suggested not to store cryptographic keys in RAM for a long time. Nevertheless, people continued to store all data in RAM carefree, including encryption keys. In 2008, encryption keys were then successfully recovered from RAM by Halderman et al. [5]. For the first time, Halderman et al. put a focus on breaking disk encryption by exploiting the remanence effect. To this end, they rebooted a running target PC by pressing its reset button, a so-called *cold boot*, and loaded a mini OS from a USB thumb drive to retrieve what is left in memory. As a result, Halderman et al. were able to break popular encryption solutions like BitLocker and FileVault. A recent study "on the practicability of cold boot attacks" [6] confirms the results by Halderman et al. In 2012, cold boot attacks were developed further by Müller and Spreitzenbarth [7] who attacked the encryption scheme of smartphones, in particular the encryption of Android-driven ARM devices. They called their data recovery tool FROST (*forensic recovery of scrambled telephones*).

The fact that cryptographic keys in main memory are unsafe has been known for almost two decades. 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.

A possibility to overcome the threat of cold boot attacks while keeping keys in main memory is to detain adversaries from accessing RAM contents. 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 recently 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.

### A. Contributions

The contributions of our work are as follows:

- With ARMORED, we provide the first *CPU-bound encryption system* for the ARM microarchitecture. ARMORED runs the AES cipher [8] on ARM processors without involving RAM. The key, the key schedule, and all intermediate values of AES are entirely stored solely in CPU registers.
  In the academic world, processor bound implementations of AES are widely accepted as a protection mechanism to cold boot attacks [9], [10], [11], [12]. However, all solutions to date have been developed for PCs based on the x86 architecture. Due to the missing AES-NI instruction set [13], which is available on CPUs from Intel, it has not been clear yet if such an implementation is possible for ARM smartphones, too.
- We implemented AES entirely on the microprocessor without a dedicated instruction set like AES-NI. We solved this task by implementing AES with Gladman's method [14] in ARM assembly language, and by exploiting ARM's multimedia register set NEON [15]. Nevertheless, we often have to recompute intermediate

values of AES for new input blocks, in particular AES round keys, because we avoid storing them in memory. The secret key is kept persistently inside debug registers of the ARM architecture during the entire runtime of a smartphone (after a short bootstrapping phase where that key is read in).
- CPU-bound encryption should 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 third party apps for Android, but requires the underlying OS to be modified. The installation and setup procedure currently require expert knowledge, such that we consider our solution as proof-of-concept code.
- Finally, we provide a security and performance analysis of ARMORED. We prove that no critical state of AES ever enters memory by observing a smartphone's RAM at runtime. On the downside, ARMORED runs twice as slow as Android's ordinary disk encryption. However, we find this drawback acceptable for many practical use cases when comparing it to the gain in security.

We provide our proof-of-concept code for ARMORED publicly under an open source license (GPL [16]) at `www1.cs.fau.de/armored`.

### B. Related Work

CPU-bound implementations of AES for the resistance of disk encryption against cold boot attacks are a well studied method in the academic world. 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 ideas of the first ever mentioned CPU-bound encryption system named *FrozenCache* [9] were illustrated. 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* [17]. 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 [18] 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* [13] (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.

### C. Outline

In Sect. II, we provide necessary background information about disk encryption in Android 4.0 and about cold boot attacks by means of FROST. In Sect. III, we describe technical details of our implementation of ARMORED. In Sect. IV, we evaluate ARMORED regarding its security, usability, correctness, and performance. And in Sect. V, we conclude with current limitations and an outlook of future applications for ARMORED.

## II. Background Information

We now provide background information about the support for full disk encryption in Android 4.0 and subsequent versions (Sect. II-A). We also give necessary information about cold boot attacks on encryption keys, in particular about FROST (Sect. II-B).

### A. Full Disk Encryption in Android 4.0

Android supports built-in full disk encryption since version 4.0, aka *Ice Cream Sandwich* (ICS). While third party apps for Android are written in Java, disk encryption resides in system space and is consequently written in C. Android is essentially based on Linux, and thus Android's encryption is based on Linux' disk encryption *dm-crypt* [19]. Dm-crypt relies on the *device-mapper* infrastructure and the *Crypto-API* that can map arbitrary devices as crypto devices. Writing to a mapped device gets encrypted and reading from it gets decrypted. As we describe in Sect. III, we provide ARMORED as dm-crypt compatible module for ARM-based Linux kernels. Basically, ARMORED is not limited to Android but can be run on ordinary Linux kernels for ARM as well.

Although dm-crypt is suitable for whole disk encryption, in Android it is not used to encrypt whole disks but the user partition only (which is mounted at `/data`). This partition is encrypted by means of the mode `aes-cbc-essiv:sha256` with 128-bit keys [20]. The AES-128 *data encryption key* (DEK) is encrypted with an AES-128 *key encryption key* (KEK), which is derived from the user PIN or password through the *password-based key derivation function 2* (PBKDF2) [21].

Unlike iOS [22], which automatically activates disk encryption when a passcode is set, Android's encryption is switched off by default. Once activated, it permanently encrypts the user storage. This process cannot be undone.

In theory, if encryption is enabled user data cannot be accessed without entering the correct PIN or password – even though the internal flash memory gets soldered out and accessed independently of the system software.

Notice that encryption in Android can only be activated if PIN-locks or passwords are in use. In Android, PINs consist of 4 to 16 numeric characters, and passwords consists of 4 to 16 alphanumeric characters with at least one letter. Other screen locking mechanisms like pattern-locks and face recognition are less secure, and this is why Google forbids them in combination with encryption. Pattern-locks, for example, can be broken by *Smudge Attacks* [23], and face recognition can simply be tricked by showing a photo of the smartphone owner [24].

### B. Cold Boot Attacks with FROST

As shown by Müller and Spreitzenbarth [7], 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 contents fade 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 briefly, because smartphones usually have no reset button. The battery must be removed fast since a 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, Müller and Spreitzenbarth 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 a phone, FROST must be installed into its 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 recently released *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.

## III. ARMORED: DESIGN AND IMPLEMENTATION

We now give details about the first CPU-bound encryption system for ARM, which we call ARMORED. We implemented ARMORED mostly in ARM assembly, 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 from the processor. Note that various AES implementations for ARM exist, for example by Bernstein and Schwabe [25], 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.

### A. 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 but 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 space, 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 divided the key-sequence into 16-bit chunks; specifically, 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 future releases 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.

### B. NEON Multimedia Registers

Besides debug registers, ARMORED is based on the multimedia register set NEON [15], which is available on ARM CPUs like the Cortex-A9 series [26]. 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 real memory.

Transferring data between general purpose registers and NEON registers can be done on byte level, just like storing data in memory. Above that, 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 in future, 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, for example.

### C. 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 calls aesenc just 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 [14]. 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, was 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 had 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. IV-A). 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 (see Sect. IV-A). For more details of our implementation, please refer to the source code that is publicly available on our webpage (`http://www1.cs.fau.de/armored`). 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 an OMAP4

chip from Texas Instruments [27]. This chip is built in devices like the Samsung Galaxy Nexus.

## IV. Evaluation

We evaluated ARMORED regarding its performance (Sect. IV-A), usability (Sect. IV-B), correctness (Sect. IV-C), and security (Sect. IV-D).

### A. 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 [17] 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 from an encrypted RAM disk.)

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 [28] (*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.

### B. Usability

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 via 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. IV-D), 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. In future, 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 yet, we believe it is practical, and we believe it would make processor bound encryption available to a large number of end-users.

### C. Correctness

To prove the cryptographic correctness of ARMORED, we used the official test vectors for AES that are listed in FIPS-197 [8]. ARMORED is integrated into the Crypto-API in a way that the Linux 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.

### D. Security

We evaluated ARMORED's resistance against (1) cold boot attacks, and (2) against other attacks such as local privilege escalations. We describe each of them in the following.

*1) Cold Boot Attacks:* To evaluate ARMORED's resistance against cold boot attacks, we reproduced cold boot attacks against Samsung Galaxy Nexus devices with FROST (see Sect. II-B). 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 upon 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 it 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 [29]. 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.

*2) Other Attacks:* 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* [30]. 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, we believe that the desired kernel integrity property can be guaranteed up to a certain degree, contrary to ordinary x86 PCs, 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, however, is disabled in Android kernels 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 [30]: "Using this [DMA] capability, we demonstrate that an attacker can expose a CPU-bound encryption key by injecting a small piece of code into the operating system kernel. This code transfers the encryption key from the CPU into RAM, from which it can be accessed using a standard DMA transfer." DMA ports like FireWire [31], [32] and Thunderbolt [33], [34] allow compromising the system space and violating kernel integrity. However, DMA ports are commonly not available on smartphones. Virtually all smartphones have USB ports for data transfers but no FireWire or Thunderbolt port. USB ports are not DMA capable.
- JTAG interfaces: Today, Android-based smartphones often have a built-in and automatically enabled JTAG interface. JTAG (*joint test action group*) is a debug port for microprocessors of embedded systems. It allows external debuggers to communicate with the embedded chip in order to set break- and watchpoints. For attackers, the JTAG interface on ARM CPUs might become the counterpart to DMA ports on x86 PCs. JTAG can break kernel integrity with physical access. That is why smartphone vendors should disable it in future releases of their mass market products. The JTAG interface is a security gap for consumer products [35].

To sum up, CPU-bound encryption protects against cold boot attacks but not against many other practical attacks. 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, however, that Android-based smartphones without 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.)

## V. Conclusions and Outlook

Concluding, as it has recently been proven by FROST, cold boot attacks enable adversaries with physical access to a phone to extract some or all of its data, even if encryption is enabled. With ARMORED, we give a concrete prototype implementation on how a countermeasure against cold boot attacks can look like on ARM. ARMORED performs encryption solely on CPU registers, thus thwarting attempts to reveal sensitive key material from RAM. Therefore, ARMORED can be classified as "Anti-FROST" as it prevents cold boot attacks on the encryption key.

### A. Limitations

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 to an adversary. So the most notable limitation is that ARMORED cannot protect information other than disk encryption keys, but other CPU-bound encryption systems face the same limitation.

Another limitation is, as stated above, that ARMORED is insecure regarding attacks via the JTAG interface. Tools like the RiffBox [36] could be used in future to defeat ARMORED, just as TRESOR can be defeated by DMA attacks, because in that case the kernel integrity property is not given.

Another notable limitation is the performance of ARMORED. The advantage of ARMORED, that it runs on all

ARM CPUs with a NEON instruction set, comes at the cost of encryption speed. However, in future we want to support dedicated crypto instructions and hardware accelerators of recent 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.

### B. Outlook

To defeat cold boot attacks, systems like ARMORED must be integrated into future 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 securely be handled by Android's PIN prompt in the future.

In comparison to its PC-based counterpart TRESOR, ARMORED inherits the following advantages from Android:

- Single keys: A point of criticism against TRESOR is that it supports single encryption keys only. Encrypting two partitions with two different keys is impossible because debug registers cannot store a second key. In Android, however, there is just a single encrypted user partition.
- Suspend-to-RAM: Another problem in TRESOR is to handle the ACPI mode S3 (suspend-to-RAM). Since CPUs are switched off during S3, the encryption key is irretrievably lost and must be re-entered upon wakeup. On Android-based ARM devices, however, there is no sleep mode that switches off the CPU.
- Kernel integrity: As stated in Sect. IV-D, the kernel integrity property can be better guaranteed on Android devices, because root accounts are disabled and DMA ports are not present. (However, the weak point of many devices today is the JTAG port which is often unnecessarily present.)
- Official releases: It is presumably impossible to get TRESOR into the Linux mainline because it violates the intended use of debug registers and breaks compatibility with existing applications like debuggers. Android, on the other hand, maintains a custom fork of the Linux kernel and does not have ambitions of strict backwards compatibility.
- Business competition: Apple's iOS is presumably secure against cold boot attacks. We did not investigate the case of Apple further, but the following statement from Apple lets us believe so: "Shortly after the user locks a device [..] the decrypted class key is discarded, rendering all data in this class inaccessible until the user enters the passcode again" [22]. That could mean Apple wipes the key from RAM each time the screen gets locked and re-derives it from the PIN only when the screen gets unlocked.

To sum up, we believe that Android can become the first platform where CPU-bound encryption is widely deployed in practice. Cold boot resistant implementations for mobile Android devices are more meaningful than for ordinary PCs. Android is the only OS today that is both completely open-source and widely in use by end-users.

## ACKNOWLEDGMENTS

We would like to thank Felix Freiling, Stefan Vömel, and Michael Gruhn for helpful comments and valuable suggestions for improving this work.

## REFERENCES

[1] S. Skorobogatov, "Data Remanence in Flash Memory Devices," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, vol. 3659, University of Cambridge, Computer Laboratory. Springer, 2005, pp. 339–353.

[2] 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 *21st USENIX Security Symposium*, UMass Amherst. Bellevue, WA: USENIX Association, Aug. 2012.

[3] R. Anderson and M. Kuhn, "Tamper Resistance: A Cautionary Note," in *Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, ser. WOEC'96. Oakland, CA: USENIX Association, 1996, pp. 1–1.

[4] P. Gutmann, "Data Remanence in Semiconductor Devices," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. Washington, D.C.: USENIX Association, 2001, pp. 4–4.

[5] 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 *Proceedings of the 17th USENIX Security Symposium*, Princeton University. San Jose, CA: USENIX Association, Aug. 2008, pp. 45–60.

[6] M. Gruhn and T. Müller, "On the Practicability of Cold Boot Attacks," in *The Eight International Workshop on Frontiers in Availability, Reliability and Security (FARES 2013)*. Regensburg, Germany: Friedrich-Alexander University of Erlangen-Nuremberg, Sep. 2013.

[7] T. Müller and M. Spreitzenbarth, "FROST: Forensic Recovery Of Scrambled Telephones," in *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ser. ACNS 2013, Banff, Alberta, Canada, Jun. 2013, http://www1.cs.fau.de/frost.

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

[9] J. Pabel, "FrozenCache – Mitigating Cold-Boot Attacks for Full-Disk-Encryption Software," in *27th Chaos Communication Congress (27c3)*. Berlin, Germany: Chaos Communication Club (CCC), Dec. 2009.

[10] T. Müller, F. Freiling, and A. Dewald, "TRESOR Runs Encryption Securely Outside RAM," in *20th USENIX Security Symposium*, University of Erlangen-Nuremberg. San Francisco, California: USENIX Association, Aug. 2011, pp. 17–17.

[11] P. Simmons, "Security Through Amnesia: A Software-based Solution to the Cold Boot Attack on Disk Encryption," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. Orlando, Florida: ACM, 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 *10th International Conference on Applied Cryptography and Network Security (ACNS '12)*, University of Erlangen-Nuremberg. Singapore: Springer-Verlag, Jun. 2012.

[13] S. Gueron, "Intel's New AES Instructions for Enhanced Performance and Security," in *Fast Software Encryption (FSE), 16th International Workshop*, Intel Corporation, Israel Development Center, Haifa. Leuven, Belgium: Springer, Feb. 2009, pp. 51–66.

[14] B. Gladman, "A Specification for Rijndael, the AES Algorithm," Webpage: http://www.gladman.me.uk, Aug. 2007.

[15] ARM Ltd., "NEON SIMD Instruction Set," Jun. 2012, http://www.arm.com/products/processors/technologies/neon.php.

[16] R. Stallman and J. Cohen, "GNU General Public License Version 2," Jun. 1991, Free Software Foundation.

[17] T. Müller, A. Dewald, and F. Freiling, "AESSE: A Cold-Boot Resistant Implementation of AES," in *Proceedings of the Third European Workshop on System Security (EUROSEC)*, RWTH Aachen / Mannheim University. Paris, France: ACM, Apr. 2010, pp. 42–47.

[18] S. Thakkar and T. Huff, "The Internet Streaming SIMD Extensions," *IEEE Computer*, vol. 32, no. 12, pp. 26–34, Apr. 1999, Intel Corporation.

[19] C. Saout, "dm-crypt: a device-mapper crypto target," 2006, http://www.saout.de/misc/dm-crypt/.

[20] Android Open Source Project (AOSP), "Notes on the implementation of encryption in Android 3.0," 2011, source.android.com/tech/encryption/.

[21] 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., Dec. 2010.

[22] Apple Inc., "iOS Security," May 2012, Whitepaper.

[23] A. J. Aviv, K. Gibson, E. Mossop, M. Blaze, and J. M. Smith, "Smudge Attacks on Smartphone Touch Screens," in *WOOT '10 (4th USENIX Workshop on Offensive Technologies)*. Department of Computer and Information Science, University of Pennsylvania, Aug. 2010.

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

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

[26] ARM, *Cortex-A9: Technical Reference Manual*, Revision: r4p0 ed., Mar. 2012.

[27] Texas Instruments Incorporated, *OMAP 4 mobile applications platform*, 2009.

[28] *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, Oct. 2010.

[29] Joe Sylve, "LiME - Linux Memory Extractor," in *ShmooCon '12*. Washington, D.C.: Digital Forensics Solutions, LLC, Jan. 2012.

[30] E.-O. Blass and W. Robertson, "TRESOR-HUNT: Attacking CPU-Bound Encryption," in *2012 Annual Computer Applications Conference*, Northeastern University, College of Computer and Information Science. Orlando, Florida: ACSAC 28, Dec. 2012.

[31] M. Becher, M. Dornseif, and C. N. Klein, "FireWire - All your memory are belong to us," in *Proceedings of the Annual CanSecWest Applied Security Conference*. Vancouver, British Columbia, Canada: Laboratory for Dependable Distributed Systems, RWTH Aachen University, 2005.

[32] P. Panholzer, "Physical Security Attacks on Windows Vista," SEC Consult Vulnerability Lab, Vienna, Tech. Rep., May 2008.

[33] Robert David Graham, "Thunderbolt: Introducing a new way to hack Macs," Feb. 2011, Errata Security.

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

[35] K. Rosenfeld and R. Karri, "Attacks and Defenses for JTAG," in *IEEE Design & Test of Computers*. Polytechnic Institute of New York University, Feb. 2010.

[36] Rocker Team, "RIFF Box JTAG Revolution," http://www.riffbox.org/, 2010.