TransCrypt: Transparent Main Memory Encryption Using a Minimal ARM Hypervisor

Julian Horsch
Fraunhofer AISEC
Garching near Munich, Germany
julian.horsch@aisec.fraunhofer.de

Manuel Huber
Fraunhofer AISEC
Garching near Munich, Germany
manuel.huber@aisec.fraunhofer.de

Sascha Wessel
Fraunhofer AISEC
Garching near Munich, Germany
sascha.wessel@aisec.fraunhofer.de

***Abstract*—Attacks on memory, revealing secrets, for example, via DMA or cold boot, are a long known problem. In this paper, we present TransCrypt, a concept for transparent and guest-agnostic, dynamic kernel and user main memory encryption using a custom minimal hypervisor. The concept utilizes the address translation features provided by hardware-based virtualization support of modern CPUs to restrict the guest to a small working set of recently accessed physical pages. The rest of the pages, which constitute the majority of memory, remain securely encrypted. Furthermore, we present a transparent and guest-agnostic mechanism for recognizing pages to be excluded from encryption to still ensure correct system functionality, for example, for pages shared with peripheral devices. The detailed evaluation using our fully functional prototype on an ARM Cortex-A15 development board running Android shows that TransCrypt is able to effectively protect secrets in memory while keeping the performance impact small. For example, the system is able to keep the E-mail account password of a typical user in the Android mail app's memory encrypted 98.99% of the time, while still reaching 81.7% and 99.8% of native performance in different benchmarks.**

***Index Terms*—RAM Encryption; Main Memory Encryption; ARM Hypervisor; Cold Boot Attack; DMA; Android**

I. Introduction

Highly mobile computing devices like smartphones are becoming omnipresent in everybody's life and are carrying a lot of private and sensitive data. This makes them valuable targets of sophisticated attacks. While protection of persistent data by means of Full Disk Encryption (FDE) is already very common, sensitive and private data in the volatile main memory, i.e. the Random Access Memory (RAM), remains unprotected. This includes, for example, classic secrets like cryptographic keys and passwords but also private data like documents, pictures and others. There are different *memory attacks* which extract main memory data from a system, such as cold boot [1]–[3] or Direct Memory Access (DMA) attacks [4]–[6]. Some of them might even be executed remotely, e.g. through a baseband processor with memory access [7].

Several existing approaches try to protect keys normally stored in RAM, e.g. by moving them from the memory into special processor registers [8]–[10] or into higher privilege levels [11]. While those approaches protect keys, they do not protect other sensitive data in RAM. There are several approaches for main memory encryption. Most of them rely on custom hardware [12]–[14] and are therefore expensive and hard to realize. Other software-based RAM encryption approaches have real-world usability issues [15], only encrypt selected processes [16], [17] or parts of selected processes [18]. Existing hypervisor-based approaches [19], [20] focus on a different attacker model and do not prevent cold boot, DMA or similar attacks from the outside.

To overcome downsides of existing approaches, we propose TransCrypt, a concept for transparent main memory encryption using a minimal hypervisor. We use hardware-supported virtualization mechanisms to transparently restrict the guest's memory access to a small and dynamically changing subset of physical RAM pages while encrypting and decrypting pages entering or leaving the set on-the-fly. This ensures that only a small part of memory containing the most recently accessed pages is unencrypted, while the major part remains securely encrypted. There are special pages, such as the ones shared with peripheral devices via DMA, for which an encryption might lead to malfunction. Hence, we introduce a mechanism to transparently detect those pages to temporarily exclude them from encryption. Being located in the hypervisor, TransCrypt has several advantages. It does not require any changes to the guest, i.e. the kernel and user space software. It is almost completely agnostic to the guest Operating System (OS) and encrypts not only user space process memory but also kernel code and data. Using a custom and minimal hypervisor TransCrypt provides a small, less error-prone code base.

Our implementation focuses on the ARM architecture [21] and the ARM Virtualization Extensions [22]. The ARM architecture is prevalent in the mobile sector and hardware virtualization is very lightweight on ARM allowing for fast and frequent switches into the hypervisor [23]. Our contributions in this paper are:

- A concept for secure hypervisor-based guest-transparent main memory encryption.
- A concept for transparent detection of special pages for which an encryption might lead to system malfunctions, such as pages used for DMA.
- A fully functional prototype implemented on an ARM Cortex-A15 multi-core development board running a full-fledged Android.
- A detailed security and performance evaluation.

 DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.232

The paper is organized as follows. Section II summarizes related work. We introduce our attacker model in Section III. Section IV gives a short overview regarding necessary hardware virtualization features. Section V describes the encryption mechanism and Section VI details how to detect special pages. Section VII explains the prototype implementation before TransCrypt is evaluated in Section VIII regarding security and performance. We conclude in Section IX.

II. Related Work

CPU-bound encryption concepts focus on removing encryption keys from RAM. Most approaches [8], [9], [11] use special CPU registers to store a key so that it is never stored in RAM. While these concepts protect keys, e.g. for FDE, they leave the rest of the RAM contents unprotected.

Hardware-based RAM encryption concepts, such as *XOM* [13], *Aegis* [14] and *CryptoPage* [12], focus on different aspects of designing secure processors with encrypted memory and encrypted memory buses. As they require changes to the hardware, they are much more expensive and harder to realize than our software-based approach. ARM TrustZone [21], [24], [25], Intel SGX [26] and AMD SME/SEV [27] are hardware security extensions of current CPUs. TrustZone and SGX are designed to allow certain small and specially designed applications to run in a secure CPU mode, a *secure enclave*, possibly from small portions of encrypted memory. As such, they do not provide a solution to the encryption of large parts of memory for normal applications and the kernel. At least TrustZone can be seen complementary to TransCrypt as it provides a means to secure the key for its memory encryption. The AMD extensions enable encryption of large parts of RAM or of specific VMs to remove trust from the hypervisor. While this achieves a similar goal as TransCrypt, the solution is AMD-only. Furthermore, AMD SEV requires changes to the hypervisor *and* the guest. Finally, all three CPU extensions might additionally bear the danger of losing key and feature control to the CPU manufacturer.

Henson et al. [15] realize RAM encryption in software on an ARM Cortex-A8 platform using a microkernel in the special and very small On-Chip RAM (OCRAM), also called iRAM, which is often provided by ARM TrustZone platforms. As it is not part of the RAM, this OCRAM is invulnerable to classic memory attacks. They dynamically swap and en-/decrypt processes between RAM and OCRAM. As the OCRAM is very small and basic features such as Memory Management Unit (MMU) support are not provided, the concept suffers from performance issues and is very hard to combine with real applications such as a normal Android OS environment.

The approaches by Chen et al. [16] and *Sentry* by Colp et al. [17] both encrypt memory of selected, "sensitive" processes. These processes are decrypted into on-SoC memory, either into locked cache (*Cache as RAM*) or into OCRAM. In case of *Sentry*, processes are encrypted when the device is suspended and decryption into cache or OCRAM is only done for processes that have to run in background. *Sentry* proposes an AES implementation to be run completely out of OCRAM. Both approaches suffer from the fact that cache locking is an optional legacy feature in newer ARM architectures [21], [25]. In contrast to TransCrypt, both approaches only encrypt selected processes and do not cover kernel space with their encryption. They require changes to the kernel or user space while TransCrypt is completely transparent. Furthermore, *Sentry* only encrypts memory when the system is suspended.

CryptKeeper by Peterson [28] introduces the concept of extending the memory hierarchy by dividing the RAM into a smaller unencrypted and a bigger encrypted part. This is somewhat similar to TransCrypt in that only a small part of recently accessed data in RAM is left unencrypted. In contrast to TransCrypt, *CryptKeeper* is realized in the Linux kernel and can therefore not protect kernel memory itself, is not transparent or agnostic to the OS and does not provide a concept for securing the encryption key.

RamCrypt [18] modifies Linux memory management to realize encryption of least-recently accessed pages. In contrast to our approach, they realize this in kernel space and activate it for special processes only. They are therefore not able to encrypt the kernel itself. Furthermore, they do not encrypt file-backed data in memory, including code and possibly confidential memory-mapped files.

Overshadow by Chen et al. [20] and the approach by Yang et al. [19] both, like TransCrypt, use encryption from a privileged hypervisor to protect memory contents. But instead of protecting the system from attacks from the outside, such as cold boot, their goal is to protect user space data from an attack from a malicious guest OS. Hence, both systems rely on storing keys or even unencrypted versions of encrypted pages [19] in RAM. This makes them highly susceptible to the outside attacks discussed in this paper and prevented by TransCrypt. Furthermore, both approaches are x86 based, specific to Linux guests, do not encrypt guest kernel data and require intercepts from the hypervisor on *every* context switch in the guest, e.g. on interrupts and syscalls. The latter is much easier to realize and causes much less relative overhead on the software-based hypervisors Overshadow and Yang et al. use for their prototypes. In a modern, hardware-assisted virtualization scenario, such as the one TransCrypt uses, those intercepts would cause a huge relative overhead.

HyperCrypt [29] is a hypervisor-based approach for RAM encryption developed at the same time as TransCrypt. In contrast to TransCrypt, the approach targets x86 and selectively focuses on server applications avoiding challenges, such as GPU support, posed by targeting a full-fledged end user device like an Android smartphone. Furthermore, HyperCrypt does not provide a dynamic, guest-transparent way to detect DMA pages. Instead it relies on paravirtualization for DMA page identification. Therefore, it requires changes to the guest as well as driver support in the hypervisor increasing its complexity and code size. Additionally, HyperCrypt does not offer a solution for efficient multi-core design and dynamic adaption of the unencrypted memory window size.

Other approaches encrypt the RAM of devices [30], [31] or groups of processes during their suspension [32] (*suspend*

to RAM). While those concepts protect suspended devices and processes, they cannot protect secrets in RAM owned by running processes.

III. Attacker Model

The primary characterization of our attacker is his ability to read parts or all of the physical main memory without involving the kernel running on the CPU. In the following, we refer to such attacks as *memory attacks*. A memory attack can be achieved using different techniques.

First, an attacker with physical access to the target device can execute a *cold boot* attack [1]–[3]. In a typical cold boot attack the attacker cools down the memory and physically moves it to another device or reboots the original device into a minimal privileged memory dump tool. Because of the remanence effect, most of the RAM contents are still intact and can be read by the attacker.

Second, an attacker can execute a DMA attack [4]–[6]. Such an attack can be characterized by the attacker controlling a peripheral device in the target to directly access the main memory via DMA without involving the CPU. Our attacker is capable of executing different DMA attacks including attacks via physical peripheral connection and via remote control of a DMA device. Examples for possible target peripherals include common devices like Graphics Processing Units (GPUs) and Network Interface Controllers (NICs) but also mobile specific targets like the baseband processor in a smartphone [7]. We assume an attacker to be able to successfully execute these attacks despite the possible presence of IOMMUs controlling the access of peripheral devices to memory, for example, because of a misconfiguration or hardware limitations.

None of the attackers is able to break cryptographic primitives. All attackers access the system from outside the CPU and are therefore not able to influence workload on the CPU. We evaluate in Section VIII-A how TransCrypt improves security against these attackers.

IV. Hardware Virtualization

Many modern CPUs provide hardware support for system virtualization. Normally, this includes a number of additions to the CPU architecture, for example, for managing guest memory, trapping privileged instructions and virtualizing devices. In the following, we shortly discuss the features that are important for TransCrypt. We focus on the ARM Virtualization Extensions [22] as our implementation is based on ARM.

The most important feature we use is the Second Level Address Translation (SLAT) which introduces an additional level of address translation for all addresses accessed by the guest. When the guest accesses a Virtual Address (VA), the guest-controlled first level translation translates the address to a *guest-physical* address, or Intermediate Physical Address (IPA) as ARM calls them. The IPA is then translated to an actual Physical Address (PA) using the SLAT controlled by the hypervisor. The SLAT is completely transparent to the guest for which it seems as if it translates VAs directly into PAs.

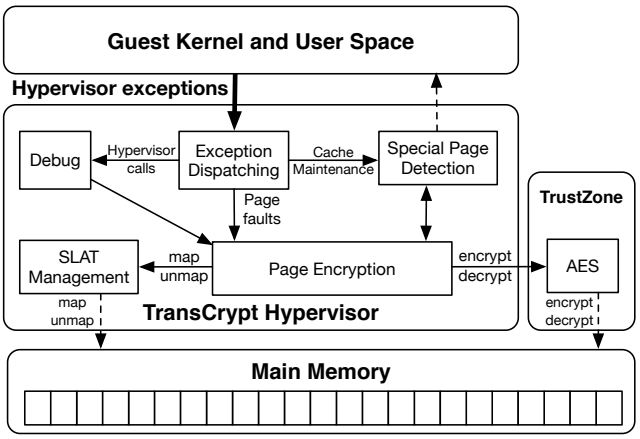

Fig. 1. Overview of the TransCrypt architecture.

This allows running multiple guests that use the same IPA space while separating them in physical memory.

Both levels of translations can define their own memory attributes and access permissions. These are then overlayed and the more restrictive configuration takes precedence. While the guest may have a valid translation for a VA, the hypervisor can still prevent access to the underlying PAs in the SLAT. This builds the basis for our memory encryption concept.

Another important feature is the ability to trap certain privileged instructions, i.e. to automatically jump into the hypervisor before executing them. As described in Section VI, TransCrypt utilizes this feature to realize a technique for recognizing memory pages used for DMA.

V. Concept and Design

TransCrypt uses the SLAT to transparently restrict the guest to a small set of unencrypted physical memory pages, the Unencrypted Page Set (UPS), while keeping the rest encrypted (with some exceptions as discussed in Section VI). TransCrypt dynamically encrypts and decrypts pages based on guest accesses and the resulting page faults.

A. Architecture

The system basically consists of the guest running in kernel and user space and TransCrypt running in hypervisor mode and partly in a secure enclave mode such as the ARM TrustZone to secure the encryption key as described in Section V-D. TransCrypt is designed as a minimal, custom hypervisor which only supports SLAT management, runs only one guest and does not virtualize any devices. This is not a conceptual limitation but drastically reduces the hypervisor's complexity making it less error-prone. The architecture of TransCrypt is depicted in Figure 1. Exceptions from the guest are initially handled in the *Exception Dispatching* module. Hypervisor Calls (HVCs) are only included for debugging purposes and are forwarded to a *Debug* module which allows analyzing page statistics and changing the size of the UPS. Traps of certain cache maintenance operations are forwarded to the *Special Page Detection* module where they are used to determine if a

page is used for DMA and must therefore be left unencrypted as detailed in Section VI-B. Page faults are the most frequent and important exceptions handled by TransCrypt. They are forwarded to the *Page Encryption* module which uses the Special Page Detection module to determine if a page should be encrypted and uses the *SLAT Management* and *AES* modules to map and encrypt pages transparently to the guest. The AES module is located in the TrustZone or a comparable Trusted Execution Environment (TEE) which allows for secure key storage as discussed in Section V-D.

B. Definitions and Initialization

As in Section IV, the guest controls its own translation from VAs to IPAs for accessing memory while TransCrypt controls the SLAT from IPAs to PAs overlaying memory attributes given by the guest. Before the guest runs, TransCrypt reserves memory for its own operation and makes it inaccessible to the guest. For the memory allocated to the guest in form of N pages, we define two basic sets of pages:

Mapped Page Set (MPS). This set contains all pages which are currently mapped in the SLAT and therefore unencrypted and available to the guest without further intervention by TransCrypt.

Special Page Set (SPS). This is the set which contains all pages that have been identified as *special* which means that they are currently not encrypted or eligible for encryption, such as, DMA pages. Details regarding special pages are discussed in Section VI.

Together, the sets contain all unencrypted guest pages. Hence, their union constitutes the Unencrypted Page Set (UPS):

$$\text{MPS} \cup \text{SPS} = \text{UPS}$$

All pages which are not part of the UPS are encrypted. We define a maximum size M for the MPS. M is the maximum number of pages accessible to the guest at any time.

TransCrypt starts with all pages unmapped ($|\text{MPS}| = 0$) and unencrypted ($|\text{UPS}| = N$). Therefore, by definition, all pages are special ($|\text{SPS}| = N$) before being evaluated and mapped for the first time.

C. Basic Mechanism

After initialization, TransCrypt gives control to the guest. Since the MPS is initialized to contain no pages, this immediately leads to an instruction page fault. The following steps are repeatedly executed to realize the memory encryption:

1) The guest accesses a page $p_{in} \notin$ MPS either by executing it or by reading or writing data from or to it.
2) The access triggers a page fault into TransCrypt. If $p_{in} \notin$ SPS, TransCrypt decrypts it.
3) TransCrypt determines if p_{in} is special (see Section VI) in which case it adds it to the SPS. Then it maps p_{in} to the guest, i.e. adds it to the MPS.
4) If the MPS does not exceed its maximum size, execution is returned to the guest. Otherwise, i.e. if $|\text{MPS}| > M$, a page $p_{out} \in$ MPS is selected for eviction using a specific eviction mechanism discussed in the following.
5) If $p_{out} \notin$ SPS TransCrypt encrypts it. Then p_{out} is unmapped, i.e. removed from the MPS, and execution is returned to the guest.

There is no differentiation between executable pages and data pages and both are equal candidates for encryption. All steps are completely transparent to the guest and do not require any explicit commands or support by the guest. The concept is therefore completely agnostic to the guest OS.

As mentioned before, if the MPS exceeds its maximum size M, a page must be evicted. Since the guest is able to access pages in the MPS without further interaction with TransCrypt, we do *not* have actual information about which page was accessed least recently and would therefore be the first candidate for eviction. The next best eviction candidate is the page that we least recently mapped to the guest. We therefore introduce an eviction algorithm we call Least Recently Mapped (LRM). Section VII discusses how we implement this algorithm with constant complexity. Figure 2 illustrates the basic encryption mechanism and its relation to the LRM page eviction. The top part of the figure shows the guest-controlled translation from VAs to IPAs and the active mappings in form of arrows to pages in IPA space. On the bottom, the figure shows the TransCrypt-controlled translation from IPAs to PAs. The figure depicts the system just at the beginning of handling a SLAT page fault in which the most recently accessed page is decrypted and mapped while the page which least recently entered the MPS is encrypted and unmapped, i.e. removed from MPS. In the example, recent accesses by the guest added three pages to the MPS which are therefore also unencrypted ($|\text{MPS}| = M = 3$). Furthermore, three pages are mapped in the guest but not part of the MPS, the normal state for most of the pages in the system. One of these pages is part of the SPS ($|\text{SPS}| = 1; |\text{UPS}| = 4$) and therefore unencrypted despite the fact that it is not part of the MPS and that it can currently not even be accessed by the guest. Such a state is important, for example, for DMA pages as discussed in Section VI. As soon as the page is accessed again by the guest, its special status is reevaluated. This is important since the page could be repurposed by the guest kernel as normal, non-DMA memory.

D. Page Encryption Parameters

For the actual encryption of a page, several aspects are to be considered. This includes key management, algorithm and mode choice as well as Initialization Vector (IV) generation.

1) Key Management: The key used for the actual page encryption *cannot* be located in memory itself, because we assume an attacker to be able to access all physical memory (see Section III). To solve this problem, our concept relies on a secure enclave such as the ARM TrustZone [21], [24], [25] where we can securely execute cryptographic operations and store the key during runtime, either in secure OCRAM or in a cryptographic coprocessor. This is not a real conceptual constraint since TrustZone is almost always present in current ARM application processors and OCRAM is a crucial

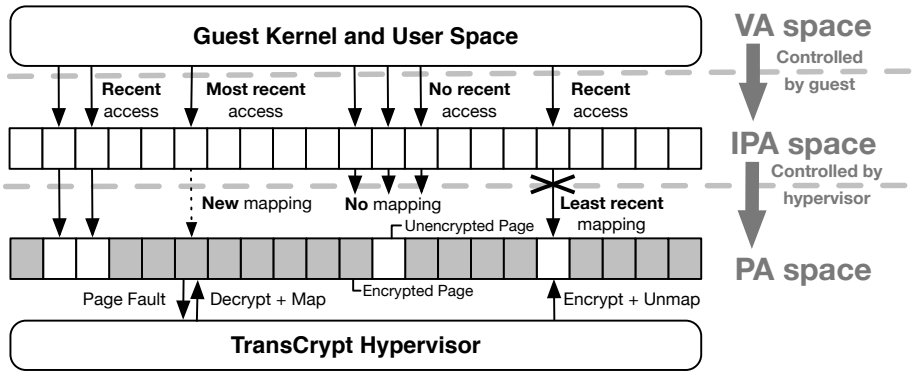

Fig. 2. Page encryption mechanism and its relation to address space translations.

component to all TrustZone platforms. For systems where no crypto coprocessor is available, Colp et al. describe how to securely implement AES [17] in software only using OCRAM. Since RAM data is not persistent, the key can be randomly generated at each system reset. Hence, a concept for persistent key storage is not necessary. If, despite being unlikely, a TrustZone-based solution is not possible on a target platform, we can store the TransCrypt key using *CPU-bound encryption* schemes [8]–[11], [18] as introduced in Section II.

2) Encryption Algorithm: The pages are encrypted symmetrically with AES. To prevent leaking information about the encrypted data, same content blocks must not result in the same ciphertext blocks. Hence, an encryption mode with IV such as Cipher Block Chaining (CBC) must be used. Page encryption runs exclusively and uninterruptible on the core that accessed the page (see Section V-F), so not much performance can be gained by choosing a highly parallelizable mode.

3) IV Generation: Each page must be encrypted using a different IV to ensure that same content pages result in different encrypted pages. An obvious choice for the IV is the physical page base address.

E. Unencrypted Page Set Size

The maximum size M of the MPS is the main configurable value in the TransCrypt concept. Increasing M increases the number of pages the guest may access at the same time and therefore also increases the size of the UPS. This, in turn, decreases security of the system as more physical pages are unencrypted anytime. Hence, M acts as a configurable trade-off between performance and security and allows fine tuning the system to the needs of the specific application. The actual impact of varying values of M is evaluated in Section VIII on the basis of our prototype implementation. Note that the size of the SPS is *not* configurable since the number of special pages is a system property and depends on the platform's hardware and OS, e.g. on how many pages are used for DMA with the GPU. We propose two ways for configuring M:

1) Static MPS Size: For this approach, M is predetermined and fixed to a certain value which fits the system's needs, especially regarding the acceptable performance overhead.

2) Dynamic MPS Size: As discussed in Section III, TransCrypt is a defense against attackers who are not or only partly able to influence the workload of the system. Furthermore, most systems spend most time in idle states. Hence, we propose a mechanism for dynamically adapting M during runtime to balance the performance loss during higher workloads by reducing the encrypted part of the memory. For that, TransCrypt can observe the time offset between SLAT page faults and adapt M to reach a specific page fault frequency. Additionally, the system should still provide a configurable hard upper bound for M to make sure that a certain amount of memory is always encrypted independent from the workload. In page fault i occurring at time t_i the new MPS size M_{i+1} can be dynamically calculated from the current size M_i using the following formula:

$$M_{i+1} \coloneqq M_i + C\left(\frac{1}{f} - \frac{t_i - t_{i-m}}{m}\right)$$

In the formula, f [1/s] denotes the desired page fault frequency and C [1/s] configures the number of pages by which M is increased or decreased for each second the difference between two consecutive page faults deviates from the desired frequency. The configurable constant value m determines how sensitive the system reacts to quick changes in the workload.

F. Multi-Core Systems

On a multi-core system, all cores must have the same view on physical memory. It must be ensured that cores only have access to unencrypted pages to not corrupt the guest. This means that all cores must share one SLAT table. Therefore, the MPS, SPS and UPS are all shared between the cores and when, for example, a core adds a page to the MPS it can be accessed by all other cores. As a result, synchronization mechanisms between the cores are necessary. As these can significantly impact performance, they must be designed carefully.

On the one hand, synchronization is required for changing state or content of pages, for example, when encrypting a page. On the other hand, synchronization is required for keeping track of the MPS in order to realize page eviction. To almost

```
1:  State:
        MPS_c: Set of pages globally mapped by core c
        SPS: Set of special pages
2:  Input:
        p_in: The page the fault was triggered on
        c: The ID of the core the fault was triggered on
3:  procedure HANDLE PAGE FAULT(p_in, c)
4:      Acquire lock for p_in
5:      if p_in ∉ SPS then
6:          Decrypt p_in
7:      else
8:          Remove p_in from SPS
9:      end if
10:     if p_in is special (see Section VI) then
11:         Add p_in to SPS
12:     end if
13:     Add p_in to MPS_c
14:     Release lock for p_in
15:     if |MPS_c| ≤ M/Total cores then return end if
16:     Get LRM page p_out from MPS_c
17:     Acquire lock for p_out
18:     if p_out ∉ SPS then Encrypt p_out end if
19:     Release lock for p_out
20: end procedure
```

Fig. 3. Memory encryption page fault handler.

eliminate all lock contesting between cores, we propse the following scheme. We introduce fine granular locks, one for each physical page, which a core must acquire to change the page's content or state. Furthermore, we split the responsibility for the MPS between the cores, which allows us to avoid *one* global and therefore contested lock. Then, each core can fill a maximum of $M/\text{Number of cores}$ page slots into the MPS. For core i, we call the resulting subset MPS_i. A core can still access *all* pages in the MPS, also the ones mapped by other cores. A page in any MPS_i will *not* trigger another page fault on *any* core until evicted. As a result, the different MPS_i are *always* disjoint. As soon as the core filled its MPS_i slots, it starts evicting pages even if other cores still have space in their MPS part. For the eviction, the core only chooses from its own MPS_i for which it does not need a lock. Figure 3 summarizes the encryption concept including multi-core handling as a pseudo code SLAT page fault handler implementation.

VI. Special Pages

We define *special pages* as pages in the physical address map of a system for which an encryption might lead to malfunction of the system. We basically identified two physical memory types that fall into this category: Memory-mapped devices, i.e. *device memory*, and memory shared with devices, i.e. DMA memory. Recognizing these and adding them to the SPS (see Section V-B) to temporarily exclude them from encryption is crucial to ensure system functionality.

A. Device Memory

Not all addresses in the physical address space of a typical system refer to main memory. Some parts of the address space are not mapped at all and other parts are used for mapping device registers for communicating with peripheral devices via memory-mapped I/O. For these memory regions, an encryption is not only useless but would also lead to system malfunctions. We identified two methods for handling device memory. First, we can analyze the guest-controlled address translation to recognize these pages. A page is special when the guest maps it as *device memory* type, making it, for example, non-cacheable. An advantage of this dynamic approach is that it does not require any prior knowledge about the location of device memory in the physical address map of the system.

The other approach is to statically analyze the physical address map of the system before runtime and simply generate fixed and always active mappings for all memory regions that do not refer to main memory. This solution has performance advantages compared to the dynamic approach. It also has the advantage that non-memory pages are completely removed from all operations and the page sets described in the previous section and do not clutter the SPS. We combine both approaches in our prototype as described in Section VII-C.

B. Memory Shared with Devices

Normal memory might be shared with peripheral devices that have access to the RAM via DMA. A prime example is the system's GPU that receives large amounts of data for rendering content on the screen from the CPU via shared memory regions and DMA. When TransCrypt unmaps and encrypts DMA pages they might still be accessed by devices using DMA. As these devices are not aware of the encryption, accessing the pages will usually lead to malfunctions. Hence, it is crucial for correct system functionality that TransCrypt is able to recognize DMA pages to add them to the SPS.

While devices might be able to access main memory, they are usually unable to access the cache hierarchy of the system's CPU. Therefore, cache coherency must be handled when communicating with devices via DMA. This behavior is leveraged by the TransCrypt hypervisor to recognize DMA pages. In the following we discuss three basic types of DMA and how TransCrypt detects them from hypervisor mode:

Always Coherent. For this type of DMA, CPU and device are always guaranteed to have the same view on their shared memory. Coherent DMA uses buffers on pages which have special attributes making them *cache coherent*. They might be non-cacheable or use a special *outer shareability* attribute, which allows some devices on ARM platforms to, in fact, access CPU caches. TransCrypt is able to detect when a guest maps a page in this way by analyzing the guest-controlled translation during the respective page fault. We call this *static DMA detection* and it is illustrated in the left part of Figure 4.

CPU to Device. For this type, the CPU must make sure that the data it sends to the device is flushed out of the caches into main memory before triggering the device to read. This can either be done by using a buffer on a page with write-through cacheability or by using explicit architectural cache maintenance operations. TransCrypt detects the first with static DMA detection and the second by trapping and emulating all cache flush operations to the main memory. TransCrypt then determines the target

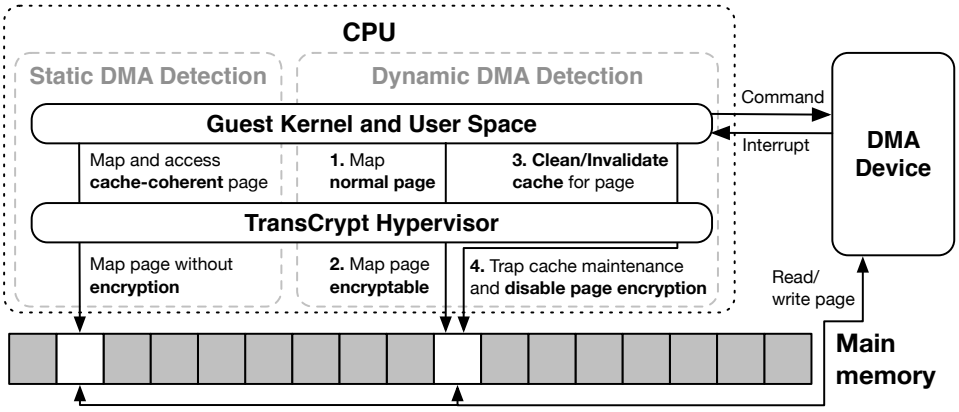

Fig. 4. Detecting DMA pages based on caching attributes and maintenance.

of the cache flush and is able to exclude the page from encryption before it is read by the device. We call this trapping-based technique *dynamic DMA detection* and it is depicted in the right part of Figure 4.

Device to CPU. For this type, the CPU must make sure to invalidate the corresponding parts of the caches before reading from a buffer written by a device to not accidentally read old cached data. For cases where the invalidation happens before triggering the device, which seems to be the normal case, it can be detected with the dynamic DMA detection described. In other cases, one has to analyze the devices and drivers in question and statically exclude the DMA pages.

Our approach has the main advantage of being agnostic and transparent to the guest software and the specific DMA devices in use. All trapped instructions are privileged so it is not possible for a guest process to "disable" encryption for certain pages. Cache maintenance operations can be trapped very selectively on ARM. This allows one to react only to operations really affecting the main memory, i.e. the "point of coherency". Additionally, modern ARM platforms, such as the one used for the prototype, allow configuring automatic cache coherency between cores, so there are normally no cache maintenance operations necessary for inter-core communication. This allows the DMA detection to specifically only target pages that are really used for communication with devices.

VII. Implementation

We implemented TransCrypt as a minimal standalone hypervisor on an Arndale board [33], a dual-core ARM Cortex-A15 developer board supporting the ARMv7 Virtualization Extensions [21], [22]. To keep the hypervisor as small as possible and reduce error-proneness, our implementation only includes functions absolutely necessary for TransCrypt. Therefore, the hypervisor does not run multiple guests or virtualize devices. Our prototype is fully functional, supports multi-core operation and is able to run an unmodified Linux kernel and Android userland including display output and touchscreen input with enabled encryption. The full implementation consists of about 4000 Lines of Code (LoC) written in C and ARM assembler.

A. Initialization

After setting up exception handlers, the hypervisor initializes its own VA to PA translation as a flat mapping over the full RAM of the Arndale board. This makes it easy to access PAs from the hypervisor. The hypervisor furthermore initializes a flat mapping for the guest SLAT from IPAs to PAs but only for the part of memory allocated to the guest. The hypervisor uses 4 KiB pages which is the smallest possible size on ARM. It also enables trapping of cache maintenance operations to the main memory (point of coherency) via the `TPC` bit in the Hyp Configuration Register (HCR) for special page detection.

B. Basic Mechanism

The core of the TransCrypt prototype is the implementation of the page fault handler as described in Section V. The handler manages an array of page meta data structures, one for each physical page allocated to the guest. Our prototype supports multi-core operation. Each core keeps track of its MPS_i (see Section V-F) and the LRM eviction with a list pointing to elements of the page data structure array. We implemented the fine granular locking as described in Section V-F using a spin lock in the data structure of each page. When mapping a page, the core appends it to its MPS list. For eviction, the core just pops the first item of its list. Both operations have constant complexity, making the implementation very efficient.

When mapping a page, it must be ensured that a possible decryption is completely finished and visible before letting the guest access the page. The ARM Cortex-A15 CPU provides a Harvard style first level cache, i.e. instructions and data are cached separately. Therefore, our hypervisor must ensure that decrypted code is flushed from the first level data cache and the respective instruction cache parts are invalidated to prevent the guest from executing encrypted instructions.

When unmapping a page, it must be ensured that none of the cores still accesses the page because of a stale entry in the Translation Lookaside Buffer (TLB). The hypervisor must hence invalidate all entries associated with the unmapped IPA on all cores. Our hypervisor ensures that all cores are in the *inner shareable domain* and uses TLB maintenance operations

to broadcast an invalidation executed on one core to all cores. Unfortunately, the ARMv7 virtualization extensions do not provide invalidation operations based on IPAs. We therefore invalidate *all* guest translations on every page unmapping. Fortunately, the ARMv8 architecture [25] provides such an operation, so that this is no issue for future implementations.

The actual encryption is done with an ARM optimized software AES implementation. As our Proof of Concept (PoC) focuses on the feasibility of the hypervisor-based concept it currently does not use encryption from the TrustZone. Running on the same CPU, the only overhead imposed by such an implementation is the TrustZone context switch. As the switch itself is very lightweight [34], especially in relation to the encryption itself, such an implementation should only add negligible overhead to our evaluated prototype. As most of the overhead comes from the actual encryption (see Section VIII-B), the crypto extensions in ARMv8 [25] promise to provide significant performance gains for the future.

The prototype does not implement the dynamic MPS size adaption as described in Section V-E2 but implements functionality for changing M at runtime for evaluation purposes.

C. Special Page Detection

The PoC implementation on the Arndale board uses several different techniques to be able to exclude all special pages from encryption as described in Section VI.

First, the hypervisor generates a fixed mapping for all physical address space regions that do not refer to main memory, as discussed in Section VI-A, making them always available to the guest and excluding them from encryption. On the Arndale board, 2 GiB from `0x40000000` to `0xbfffffff` referring to main memory remain, minus the space allocated to the hypervisor for the main encryption operation.

Second, the PoC queries the guest translation for the page in question using the architectural translation registers `ATS1CP*` [21]. The hypervisor analyzes the obtained guest's memory attributes and marks a page as special if it is not normal memory with inner and outer write-back cacheability or if it is outer shareable. This realizes both, the dynamic approach for device memory discussed in Section VI-A as well as the static DMA detection (see Section VI-B) in one step.

Third, the prototype realizes dynamic DMA detection (see also Section VI-B) by trapping and emulating all cache maintenance operations that use VAs and target the main memory. For all data cache operations in this group, namely `DCIMVAC`, `DCCMVAC` and `DCCIMVAC`, the hypervisor finds the affected page, decrypts it if necessary and marks it as special.

While the described techniques catch most of the special pages, there is one platform specific exception originating in the Mali GPU driver of the guest kernel. An analysis of this driver revealed that it employs a "physical allocator" that allocates highmem pages, i.e. pages from memory normally used for user space data, and maps it into the kernel space using `kmap()`. These pages end up in the `pkmap` (persistent kernel map) VA region in the kernel VA address space. Hence, we exclude this virtual address range (2 MiB on the Arndale) from encryption. Note that the guest still runs unmodified.

With the described mechanisms, our memory encryption prototype is able to host a full, unmodified Android OS, without impairing the system's functionality.

TABLE I
PAGE STATISTICS AFTER FINISHING ANDROID BOOT WITH $M = 6000$.

	SPS		MPS		Total	
	Dyn.	Stat.	$\cap$ SPS	$\setminus$ SPS	UPS	Enc.
All	15072	1890	85	5915	22877	**69352**
Kernel	791	1810	22	1572	4173	**10706**

VIII. EVALUATION

To evaluate the security and performance of TransCrypt, we ran several experiments and benchmarks on our prototype implementation. For all experiments, we ran a multi-core Linux 3.0.31 kernel and Android 4.1.1 on the Arndale board. In most experiments, we tested with different fixed values for the maximum MPS size M. As discussed in Section V-E, M can be used to adapt the security margin as a trade-off versus the performance of the system. The smaller M, the less data is unencrypted in memory and the more recently the data must have been accessed to be still unencrypted.

A. Security

To get an impression on how the memory encryption and the special page detection behave, we collected page statistics from our prototype immediately after the guest finished booting up Android. We chose a fixed M of 6000 for the experiment. Since the Arndale board has two cores, this results in each core managing a maximum of 3000 mapped pages, as described in Section V-F. The results are summarized in Table I. The statistics include the SPS size and the number of special pages detected dynamically and statically (see Section VI). Furthermore, the statistics show how many of the MPS pages are in the SPS as well as a total of encrypted and unencrypted pages, i.e. the UPS. Additionally to the statistics over all pages ever mapped by the hypervisor, we generated data for the Linux kernel `lowmem` region to determine how actively the kernel space is encrypted. The `lowmem` is the memory region which is contiguously mapped into the kernel and stores kernel code and most of the kernel data structures.

There are several interesting results to be read from the statistics. The SPS dynamic detection is mostly active for user space pages, while the static detection is almost exclusively active for kernel pages. The high amount of special pages can be explained by the fact that the system just finished booting, which involves a higher kernel activity than during normal runtime. The MPS is almost completely filled with non-special pages, which shows that the special page detection is very specific, so that most of the current working set is considered for encryption. The summary of encrypted and unencrypted pages shows that about 75% of all pages and 72% of the kernel

TABLE II
PERCENTAGE OF TIME THE E-MAIL APP'S MEMORY CONTAINS THE PLAIN ACCOUNT PASSWORD FOR DIFFERENT M IN THE SAMPLE USE CASE (1 MIN E-MAIL APP $\rightarrow$ 1 MIN HOME SCREEN $\rightarrow$ 1 MIN BROWSER) AND FOR THE *typical user* CALCULATED BASED ON THE AVERAGE SAMPLES.

$M =$	4000	8000	**10000**	14000	18000	36000
Worst	0.66	4.35	**12.49**	33.04	38.74	73.03
Best	0.11	0.28	**0.35**	8.77	30.16	68.70
Avg.	0.34	1.14	**3.37**	16.98	34.07	70.43
Typ.	0.1	0.34	**1.01**	5.09	10.22	23.5

`lowmem` pages used during the Android boot are currently encrypted. The amount of encrypted memory can be increased further by choosing a smaller M.

To confirm that the kernel space encryption is very active, we attached a JTAG debugger to the system to simulate a memory attack. Without memory encryption, the debugger is able to read and interpret kernel data structures, such as page tables and task lists, for example, to show the processes running on the system. With memory encryption, most of this functionality stops working. While this does not yet necessarily mean that secrets are protected in memory, memory dumps are immediately much harder to analyze forensically without being able to reliably access kernel data structures.

To find out how well the system protects user space secrets, we analyzed the process memory of the Android Mail app. After initializing the app with a test E-Mail account, during normal operation the app's memory showed four occurrences of the address/password combination on three different pages. We marked the physical pages in our hypervisor tracking their encryption and decryption. While one of the pages remains encrypted all the time, the two others are decrypted when the app is brought to foreground (and checks for new mail). To quantify the time the password is unencrypted in memory, we devised the following sample use case:

1) Open the E-Mail app and wait for one minute.
2) Close the E-Mail app and wait for one minute.
3) Open the Browser app and wait for one minute.

Furthermore, we define a *typical user* who uses the E-Mail app for 3 minutes every 30 minutes (including automatic fetches) for 24 hours of the day. In 85% of the mail app uses, he opens another app such as the browser afterwards. Table II shows the percentage of time the E-Mail password is unencrypted in the app's memory for our sample test case and the *typical user* for which we calculated the percentage based on the average sample case. The password is counted as unencrypted as soon as *one* of the three pages containing it is unencrypted.

The results for the sample use case can be interpreted as follows. If the percentage is below or around 33% it means that the password is encrypted at the latest when the E-Mail app is closed. This is the case for all M except for $M = 36000$. For this case, the password is encrypted as soon as the browser is opened. Based on these observations, we can calculate the percentage for our *typical user* based on the average case test results. For example, for this user using $M = 10000$, the E-Mail password is unencrypted in memory for less than 15 minutes over the whole day. Considering that we get very good benchmark results already for $M \leq 10000$ as shown in the next Section, the results confirm that TransCrypt is able to efficiently protect real secrets in memory.

TABLE III
PERFORMANCE OF THE MEMORY ENCRYPTION PROTOTYPE AS PERCENTAGE OF NATIVE PERFORMANCE FOR DIFFERENT BENCHMARKS.

$M =$	4000	8000	**10000**	18000	36000
CoreMark	99.0	99.8	**99.8**	99.9	99.9
Antutu	60.9	77.9	**81.7**	82.6	87.4

B. Performance

We evaluated the performance of the prototype using two benchmarks. The CoreMark [35] benchmark measuring integer performance without GUI and the Antutu [36] Android app measuring overall performance including CPU, RAM, GPU and I/O speed. The averaged results are summed up in Table III, showing the TransCrypt prototype's performance as percentage of native performance without encryption.

For CoreMark, the TransCrypt performance is almost indistinguishable from the native performance. The Antutu benchmark has a much larger working data set including assets for 3D graphic and user interface and should give an impression on how TransCrypt performs running a demanding app such as a game. Despite being a challenging real-world test, the Antutu benchmark already reaches more than 80% of native performance for $M = 10000$.

Most of the performance impact is caused by the encryption itself. Our prototype uses a software AES implementation, so switching to a hardware implementation such as the AES extensions in ARMv8, promises to significantly improve performance. As described in Section VII, ARMv8 also offers a much more efficient way for SLAT TLB invalidation, which can further improve performance.

C. Discussion

Our attacker is able to execute a cold boot attack. On a normal system, this leads to a leak of all RAM data including keys, e.g. E-mails, passwords and documents. With TransCrypt, as shown for the E-mail account password, sensitive data is encrypted with high probability using a key which is stored in a location not vulnerable to cold boot.

Our attacker is able to execute different reading DMA attacks, as discussed before in our attacker model. In all those attacks, the attacking DMA device "acts on its own", meaning that the guest kernel on the CPU does not initiate the DMA access, at least not for the addresses the device is maliciously accessing. Therefore, our DMA detection mechanism described in the Section VI, does *not* exclude the attacked pages from encryption, providing security for these cases. On a normal system, a DMA memory attack leads to leakage of

all memory data. With our memory encryption, a DMA attack reads encrypted memory instead of sensitive data with a high probability as shown for the E-Mail password.

We based our evaluation on experiments with different fixed values of M to analyze its influence on the encryption probability and the performance. As shown, choosing $M = 10000$ already provides a very good trade-off between security and performance. By using a dynamic M adaption, as described in Section V-E, this trade-off can even be improved further.

IX. Conclusion

In this paper, we presented TransCrypt, a concept and implementation for guest-transparent encryption of kernel and user memory from a custom minimal hypervisor to protect against memory attacks, for example, via DMA or cold boot. We furthermore introduced a concept and implementation for detecting special pages to be excluded from the encryption, e.g. for DMA, in a transparent and guest-agnostic way on ARM architectures. We developed a fully functional prototype on the Arndale ARM Cortex-A15 development board. Our evaluation shows that the system can effectively protect secrets in memory while keeping the performance impact small. For example, the system is able to keep the E-mail account password of a typical user in the Android mail app's memory encrypted 98.99% of the time, while still reaching 81.7% and 99.8% of native performance in different benchmarks.

References

[1] 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 Encryption Keys." *USENIX Security Symposium*, 2008.

[2] P. Gutmann, "Data Remanence in Semiconductor Devices," in *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*. USENIX Association, 2001, p. 4.

[3] T. Müller and M. Spreitzenbarth, "FROST," in *Applied Cryptography and Network Security*. Springer Berlin Heidelberg, Jun. 2013.

[4] A. Boileau, "Hit by a bus: Physical access attacks with Firewire," *Presentation, Ruxcon*, 2006.

[5] M. Becher, M. Dornseif and C. N. Klein, "FireWire: All Your Memory Are Belong To Us," *Proceedings of CanSecWest*, 2005.

[6] P. Stewin and I. Bystrov, "Understanding dma malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 21–41.

[7] R.-P. Weinmann, "Baseband attacks: remote exploitation of memory corruptions in cellular protocol stacks," in *WOOT'12: Proceedings of the 6th USENIX conference on Offensive Technologies*, University of Luxembourg. USENIX Association, Aug. 2012.

[8] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR Runs Encryption Securely Outside RAM." in *USENIX Security Symposium*, 2011.

[9] J. Götzfried and T. Müller, "ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices," in *2013 Eighth International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2013.

[10] 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*. ACM, 2011.

[11] T. Müller, B. Taubmann, and F. C. Freiling, "TreVisor - OS-Independent Software-Based Full Disk Encryption Secure against Main Memory Attacks," in *Applied Cryptography and Network Security*. Springer, 2012, pp. 66–83.

[12] G. Duc and R. Keryell, "CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection," *ACSAC*, pp. 483–492, 2006.

[13] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[14] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," in *Proceedings of the 17th annual international conference on Supercomputing*. ACM, 2003, pp. 160–171.

[15] M. Henson and S. Taylor, "Beyond full disk encryption: protection on security-enhanced commodity processors," in *ACNS'13: Proceedings of the 11th international conference on Applied Cryptography and Network Security*. Springer Berlin Heidelberg, Jun. 2013.

[16] X. Chen, R. P. Dick and A. Choudhary, "Operating system controlled processor-memory bus encryption," in *Design, Automation and Test in Europe, 2008. DATE'08*. IEEE, 2008, pp. 1154–1159.

[17] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting Data on Smartphones and Tablets from Memory Attacks," in *ASPLOS '15: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, USA: ACM, Mar. 2015.

[18] J. Götzfried, T. Müller, G. Drescher, S. Nürnberger, and M. Backes, "Ramcrypt: Kernel-based address space encryption for user-mode processes," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: ACM, 2016, pp. 919–924.

[19] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: ACM, 2008.

[20] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008.

[21] ARM Limited, *ARM Architecture Reference Manual – ARMv7-A and ARMv7-R edition*, July 2012.

[22] R. Mijat and A. Nightingale, "Virtualization is Coming to a Platform Near You," ARM Limited, Tech. Rep., Jan. 2011.

[23] J. Horsch and S. Wessel, "Transparent Page-Based Kernel and User Space Execution Tracing from a Custom Minimal ARM Hypervisor," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, Aug 2015, pp. 408–417.

[24] T. Alves and D. Felton, "TrustZone: Integrated Hardware and Software Security – Enabling Trusted Computing in Embedded Systems," 2004.

[25] ARM Limited, *ARM Architecture Reference Manual – ARMv8-A, for ARMv8-A architecture profile*, 2013.

[26] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd Int. Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013.

[27] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption," AMD Inc., Tech. Rep., Apr. 2016.

[28] P. A. H. Peterson, "Cryptkeeper: Improving security with encrypted RAM," in *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*. IEEE, 2010, pp. 120–126.

[29] J. Götzfried, N. Dörr, R. Palutke, and T. Müller, "HyperCrypt: Hypervisor-Based Encryption of Kernel and User Space," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*, Aug 2016, pp. 79–87.

[30] L. Zhao and M. Mannan, "Hypnoguard: Protecting Secrets Across Sleep-wake Cycles," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 945–957.

[31] M. Huber, J. Horsch, and S. Wessel, "Protecting Suspended Devices from Memory Attacks," in *Proceedings of the 10th European Workshop on Systems Security*, ser. EuroSec'17. New York, NY, USA: ACM, 2017, pp. 10:1–10:6.

[32] M. Huber, J. Horsch, J. Ali, and S. Wessel, "Freeze & Crypt: Linux Kernel Support for Main Memory Encryption," in *14th International Conference on Security and Cryptography (SECRYPT 2017)*, INSTICC. ScitePress, 2017.

[33] Insignal Limited, "Arndale Board," http://www.arndaleboard.org.

[34] D. Zhang, "TrustFA: TrustZone-Assisted Facial Authentication on Smartphone," Tech. Rep., Dec. 2014.

[35] E. M. B. Consortium, "Coremark," http://www.eembc.org/coremark/.

[36] AnTuTu Developers, "Antutu," http://www.antutu.com/en/index.shtml.