2016

CaSE: Cache-Assisted Secure Execution on ARM Processors

Ning Zhang
Virginia Polytech Inst & State Univ, Blacksburg, VA 24061 USA;

Wenjing Lou
Virginia Polytech Inst & State Univ, Blacksburg, VA 24061 USA;

Y. Thomas Hou
Virginia Polytech Inst & State Univ, Blacksburg, VA 24061 USA;

Kun Sun
Coll William & Mary, Dept Comp Sci, Williamsburg, VA 23185 USA

Recommended Citation

Zhang, N., Sun, K., Lou, W., & Hou, Y. T. (2016, May). Case: Cache-assisted secure execution on arm processors. In 2016 IEEE Symposium on Security and Privacy (SP) (pp. 72-90). IEEE.

2016 IEEE Symposium on Security and Privacy

CaSE: Cache-Assisted Secure Execution on ARM Processors

Ning Zhang*, Kun Sun†, Wenjing Lou*, Y. Thomas Hou*
*Virginia Polytechnic Institute and State University, VA
†Department of Computer Science, College of William and Mary, Williamsburg, VA

***Abstract*—Recognizing the pressing demands to secure embedded applications, ARM TrustZone has been adopted in both academic research and commercial products to protect sensitive code and data in a privileged, isolated execution environment. However, the design of TrustZone cannot prevent physical memory disclosure attacks such as cold boot attack from gaining unrestricted read access to the sensitive contents in the dynamic random access memory (DRAM). A number of system-on-chip (SoC) bound execution solutions have been proposed to thaw the cold boot attack by storing sensitive data only in CPU registers, CPU cache or internal RAM. However, when the operating system, which is responsible for creating and maintaining the SoC-bound execution environment, is compromised, all the sensitive data is leaked.**

In this paper, we present the design and development of a cache-assisted secure execution framework, called CaSE, on ARM processors to defend against sophisticated attackers who can launch multi-vector attacks including software attacks and hardware memory disclosure attacks. CaSE utilizes TrustZone and Cache-as-RAM technique to create a cache-based isolated execution environment, which can protect both code and data of security-sensitive applications against the compromised OS and the cold boot attack. To protect the sensitive code and data against cold boot attack, applications are encrypted in memory and decrypted only within the processor for execution. The memory separation and the cache separation provided by TrustZone are used to protect the cached applications against compromised OS.

We implement a prototype of CaSE on the i.MX53 running ARM Cortex-A8 processor. The experimental results show that CaSE incurs small impacts on system performance when executing cryptographic algorithms including AES, RSA, and SHA1.

***Keywords*-TrustZone; Cache; Memory Encryption**

I. Introduction

Smart devices are playing an increasingly important role in our daily life. As the most widely deployed CPU in mobile devices, ARM family processors have been used in 4.5 billion mobile phones to process and store sensitive data [1], [2]. For instance, around 51% of U.S. adults bank online and 35% of them use mobile phones to perform online transactions [3]. Meanwhile, fueled by the lucrative black market for mobile malware, an increasing number of system vulnerabilities have been identified and exploited to compromise the mobile OS [4]. McAfee Lab reported a 24% increase in the unique number of mobile malware in Q4 2015 [5].

To enhance the security of embedded systems, ARM provides a hardware security extension named *TrustZone* to protect sensitive code and data of applications in an isolated execution environment against a potentially compromised OS [6]. TrustZone has been widely adopted not only in academic research projects [7], [8], [9], [10], [11], [12], but also in commercial products [13], [14], [15]. However, the design of TrustZone cannot prevent physical memory disclosure attacks such as cold boot attacks [16], [17], [18], [19]. Since mobile phones are frequently stolen, when attackers have physical access to the mobile devices, they can gain unrestricted access to the contents in the DRAM. Unfortunately, TrustZone does not enforce encryption of memory in the privileged environment like SGX [20], [21]. As a result, sensitive information, such as cryptographic key material, is not secured even if it is stored in TrustZone protected physical memory when adversaries have physical access to the mobile device.

To protect against physical memory disclosure attacks, SoC-bound execution solutions have been proposed to move sensitive data out of DRAM and save them in processor registers [22], [23], [24], processor cache [25], [26], [27], [18] or internal RAM [18]. All these SoC-bound execution solutions can effectively thaw physical memory attacks under a strong assumption that the OS, which is responsible for creating and maintaining the SoC-bound execution environment, can be trusted. The justification for this design assumption is that when the OS is compromised, there is no need for attackers to launch a cold boot attack, because the OS can directly access the entire DRAM. However, it is not true for ARM processors with TrustZone support. Though TrustZone can prevent a malicious OS from accessing protected secure memory, it cannot defend against cold boot attacks. Thus, it is critical to protect mobile systems against multi-vector attacks [28] including software attacks and physical memory disclosure attacks.

In this paper, we propose a cache-assisted secure execution system called *CaSE* that can protect against both software attacks and physical memory disclosure attacks on ARM-based devices. The basic idea is to create a secure environment in the CPU cache and use TrustZone to prevent the potentially compromised OS from accessing the secure environment. Thus, CaSE can protect both confidentiality and integrity of the application's code and data against both software attacks and physical memory disclosure attacks.

To protect against physical memory disclosure attacks, CaSE creates an execution environment inside the ARM pro-

DOI 10.1109/SP.2016.13

cessor by loading and executing an application completely within the CPU cache. Cache is designed to be a hardware mechanism that is transparent to the system software except for a small number of maintenance instructions. Therefore, we solve several challenges to create a cache-assisted execution environment.

First, to make computation SoC-bound, the application code, data, stack and heap have to be stored in and only in the cache. The memory for each component in the application address space has to be allocated carefully to eliminate cache contention. Unfortunately, none of the publicly available ARM documents details the mapping from memory addresses to cache line indexes. In order to correctly place and optimize application memory in the cache, we design and perform experiments to obtain cache mapping schemes of the targeted hardware platform.

Second, once the application is loaded in the cache, we make use of the hardware-assisted cache locking function to pin down portions of the cache, without significantly impacting the system performance. With the ability to control eviction policy on cache lines that store the sensitive data, it is possible to enable context switching between the protected application and the rest of the system without concerning the execution of other programs will cause eviction of the sensitive contents from cache to DRAM.

Third, since the application is still encrypted when loaded into DRAM, it needs to be decrypted completely within cache before being executed. In many processor architectures, including ARM, instruction cache and data cache are not guaranteed to be coherent. When an application decrypts its own code back into the process address space, instruction cache and data cache become incoherent. Such issue of incoherent cache caused by self-modifying programs is often resolved by flushing the cache. In CaSE, flushing the cache fails our efforts of running applications entirely inside the SoC. To solve this problem, we synchronize the incoherent data cache and instruction cache by utilizing the unified last level cache in the processor.

TrustZone is used to protect the cache-assisted isolation environment against an untrusted OS. Cache lines in TrustZone-enabled ARM processors are built with an extra *non-secure (NS)* bit to indicate whether the line belongs to the secure world or the normal world. Therefore, the rich OS in the normal world cannot access or manipulate the cache lines used by the secure world. The secret key to decrypt the application is saved in the secure world cache. Without the key, a compromised rich OS cannot decrypt the application code, which may be misused by attackers to reverse engineer proprietary algorithms or find potential vulnerabilities. CaSE offers two running modes depending on whether secure world cache or normal world cache is used to create the environment for the SoC-bound execution. These two modes provide a trade-off between the system security and the run-time performance.

We implement a prototype of CaSE on the i.MX53 running ARM Cortex-A8 processor. Using the CaSE, we show that it is possible to execute a kernel integrity checker and a suite of cryptographic algorithms including AES, RSA, and SHA1 in the cache with small performance impacts.

In summary, we make the following contributions,

- We propose a secure cache-assisted SoC-bound execution framework that can protect sensitive code and data of applications against both software attacks from a compromised rich OS and physical memory disclosure attacks that can gain unrestricted access to the DRAM.
- We present a systematic study on designing and securing our cache-assisted SoC-bound execution environment on ARM platforms. We demonstrate the applicability of our system by prototyping several popular cryptographic algorithms along with a kernel integrity checker.
- We implement a prototype on the i.MX53 running ARM Cortex-A8 processors. The experimental results show that CaSE has small impacts on the system performance.

The remainder of the paper is organized as follows. Section II introduces background knowledge. Section III presents the threat model and assumptions. The CaSE architecture is presented in Section IV. The prototype is detailed in Section V. Section VI provides discussion on the experimental results. The extensions of CaSE are discussed in VIII. Related works are presented in Section IX. Finally, Section X provides the conclusion of the paper.

II. Background

We first introduce the ARM TrustZone hardware security extension. Then we discuss the generic ARM cache architecture along with the changes in the cache design due to the addition of TrustZone.

A. ARM TrustZone

TrustZone is a set of hardware security extensions, consisting of modifications to the processor, memory, and peripherals [6]. It has been supported since ARMv6, and most of the recent ARM system-on-chip processors support this security extension. The main purpose of TrustZone is to provide an end-to-end, complete system isolation for secure code execution. The isolated environment provided by TrustZone is often referred to as the *secure world*, while the traditional operational environment is often referred to as the *normal world*, the *non-secure world*, or the *rich OS*.

Based on the world that the processor is in, different system resources can be accessed. The *security configuration register (SCR)* in the *CP15* coprocessor is one of the registers that can only be accessed while the processor is in the secure world. SCR contains an *NS (non-secure)* bit that governs the security context of the processor. When *NS* bit is cleared, the processor is in the secure world. When *NS* bit is set,

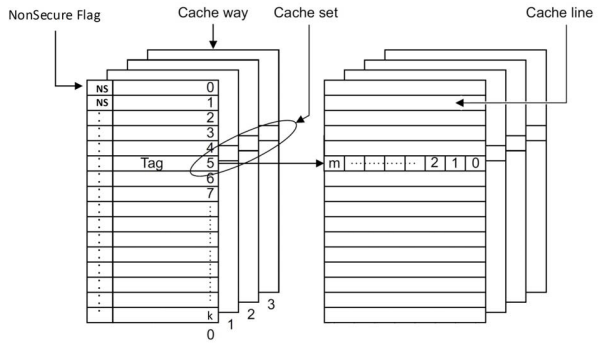

Figure 1: Cache Architecture in ARM TrustZone

the processor is in the normal world. The only exception is when the processor is in the monitor mode, which can be triggered by either interrupts or *secure monitor call (SMC)* instruction.

B. Cache Architecture in ARM Processors

Cache is considered to be the first level memory system in ARM. It is usually constructed with a fast and expensive static random access memory (SRAM). Most of the current processors have multiple levels of cache, including level one (L1) cache, level two cache (L2), and translation lookaside buffer (TLB). Modern high-end processors typically have 32 KB to 1 MB L1 cache, and the size of L2 ranges from 256 KB to 2 MB. Since cache is small compared to the total amount of addressable memory, N-way set associative table is often used to organize the cache.

A typical 4-way set associative table is shown in Figure. 1. A physical memory address is indexed into k cache lines, where k is the set size. As there are four tables of size k, the way number is four. Therefore, for any given memory address, it can be in k^{th} set entry of any way. For each cache line, there is a tag carrying the hash value of the index along with the status bits. With the introduction of TrustZone in the ARM architecture, all levels of cache have been extended with an additional *NS* tag bit, which records the security state of the transaction that accesses the memory [6]. It eliminates the need for a cache flush when switching between the two worlds, significantly improving the system performance. The content of the caches, with regard to the security state, is dynamic. Any cache line can be evicted to make space for new data, regardless of its security state. In other words, it is possible for a secure cache line fill to evict a non-secure cache line, and vice versa.

III. Threat Model and Assumptions

A. Threat Model

Sophisticated cyber attacks nowadays involve multi-stage, multi-vector attacks [28]. We assume that attackers can use both software attacks and physical memory disclosure attacks to obtain sensitive information in the DRAM.

1) Software Attack: Due to the increasing complexity of the mobile OS kernel, attackers can often exploit various kernel vulnerabilities to compromise the mobile OS. Therefore, we assume that successful software attacks can lead to the compromise of the OS and thus gain unrestricted access to not only DRAM but also the CPU cache and registers.

It is well known that an adversary can use direct memory access (DMA) attacks [29] to gain arbitrary access to physical memory on desktop computers through DMA channels such as FireWire, Thunderbolt, and PCI Express. Though DMA ports are not commonly available on current mobile devices and USB ports are not DMA capable, it is still possible for the attacker to misuse built-in DMA capable I/O devices such as LCD controller and storage controller [4]. Therefore, we consider DMA attack as an attack vector available to the compromised OS.

2) Physical Memory Disclosure Attack: With physical access to the mobile devices, there are many types of physical attacks, and it is hard to anticipate all of them. For example, if the JTAG interface is enabled on production systems, the attacker can connect a JTAG debugger to manipulate system states of the normal world. Fortunately, the secure world is protected from JTAG with the built-in protections from TrustZone. Instead, the attacker can use other advanced hardware to examine SoC internals or change DRAM state [30].

In this work, we focus on physical memory disclosure attacks, such as cold boot attacks [16], [17], [31], which exploit the remanence effect of physical memory to gain unrestricted read access to system memory. In general, there are two types of cold boot attacks: (1) resetting the computer to load a malicious OS from the attacker, and (2) unplugging and placing DRAM chips into another machine controlled by the attacker. Moreover, attackers can use bus snooping attacks [32] to capture the sensitive data when it is being loaded from or written to the DRAM. Note that though TrustZone can be used to protect secure code execution against the compromised rich OS in the normal world, the DRAM used by the secure world is still vulnerable to a physical memory disclosure attack since no encryption is enforced on the DRAM.

B. Assumptions

We assume the ARM platform supports the TrustZone hardware security extension. The high assurance boot (HAB) and system isolation between the two worlds provided by TrustZone can be trusted. We assume the secure application running in the secure world can be trusted and will not leak its information deliberately. The attackers can launch various software attacks and physical memory disclosure attacks in order to freely access the sensitive data in DRAM memory. Moreover, after gaining the root privilege in the normal world through software attacks, the attacker can also access the CPU cache and registers of the normal world

inside the processor. However, she will not be able to access the processor cache or registers in the secure world due to the protection of TrustZone. We assume that attackers with physical access to the mobile devices cannot utilize sophisticated hardware to access the SoC-bound data in cache or registers. Side channel attacks such as timing and power analysis are out of the scope of this paper.

IV. CaSE Architecture

CaSE is designed to provide a secure and isolated SoC-bound execution using the commodity hardware components running ARM processors. We first present our security goals and then give a system overview which focuses on how these goals are achieved in CaSE.

A. Security Goals

To protect against both software attacks and physical memory disclosure attacks, we design CaSE to satisfy the following security goals:

1) SoC-bound Execution Environment: The computation and memory of the application shall be within the physical boundary of the SoC. Since physical memory disclosure attacks are capable of revealing all memory contents outside the SoC, CaSE needs to use the memory that is within the physical boundary of the SoC, such as on-chip memory or processor cache to create a SoC-bound execution environment.

2) Isolated Execution Environment: The system shall be able to provide an isolated execution environment. In other words, it shall be able to bootstrap and maintain an execution environment that is completely isolated from the compromised mobile OS, including separation for processor, memory, and peripherals. On ARM processors, TrustZone can be used to achieve this goal.

3) Memory Protection Outside the Execution Environment: To protect both integrity and confidentiality of application code and data, all program information outside the physical boundary of the SoC shall be protected by cryptography. More specifically, code and data of the application shall be encrypted when they are saved into external DRAM due to memory paging, context switch, etc.

B. CaSE Overview

The overall system architecture is shown in Figure. 2. Cold boot attackers can gain unrestricted read access to all external DRAM, including those used by the system as either the secure world memory or the normal world memory. On the other hand, software attacks allow adversaries to access and manipulate memory contents of the normal world. The protected application is encrypted in the DRAM to ensure its confidentiality. When a user invokes an application, the CaSE controller will first load the encrypted application into the L2 unified cache. Then CaSE controller verifies and decrypts the application completely within cache and sets

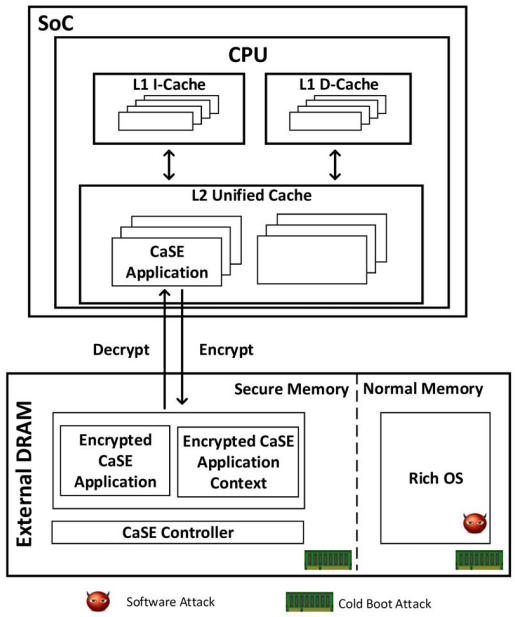

Figure 2: System Architecture

up the execution environment with cached memory. Using the hardware-assisted memory protection by TrustZone, the cache-based execution environment is isolated from software attacks from the rich OS in the normal world. Lastly, the application context is encrypted before written to memory such that sensitive information never leaves the SoC in plain text.

By executing applications only in the cache of an isolated environment provided by TrustZone, CaSE can defend against both software attacks that compromise the OS in the normal world and physical memory disclosure attacks such as cold boot attacks.

C. Constructing the SoC-bound Execution Environment

SoC-bound execution ensures that the execution of a piece of code is entirely enclosed within the physical boundary of the SoC. More specifically, the code, data, stack, and heap of the application should all be allocated to the CPU cache. Therefore, cold boot attacks cannot read either the program state or the program itself. To enable a SoC-bound execution in CPU cache, we need to solve several key challenges.

First, none of the publicly available ARM documents describes the mapping from physical memory address to cache line index in the cache way. We have to design and perform experiments to figure out this mapping for both L1 and L2 caches in ARM processors. Our results indicate that the cache organization of Cortex-A8 is similar to many other platforms in x86 systems [33], [34]. Second, since there is no direct access to cache lines from system software, we need to develop a method to precisely load memory into cache lines and avoid cache eviction during the load, run

and exit stages of the application. Third, when processor cache is used to store both code and data, self-modifying programs can cause cache incoherency between Instruction Cache (I-Cache) and Data Cache (D-Cache) in the first level cache. We solve this problem by redirecting memory write to the second level unified cache, where the cache lines are used for both instruction and data.

By tackling the three challenges above, our SoC-bound execution can load both code and data into L2 cache and protect the confidentiality of code and data against cold boot attack. However, a compromised OS from software attacks can still access the contents of the CPU cache. Therefore, CaSE also needs to isolate the SoC-bound execution from the compromised OS.

D. Isolating the SoC-bound Execution from Rich OS

TrustZone provides an *NS* flag in each cache line indicating its security state. Based on the security context of the system, CPU cache is marked as either secure or normal. We call the cache lines used by the secure world *secure cache* and the ones used by the normal world *normal cache*. TrustZone can ensure that the rich OS in the normal world cannot access the secure cache. Thus, a straightforward solution is to use secure cache to create the SoC-bound execution environment, as shown in Figure. 2. Alternatively, it is possible to use normal cache to protect sensitive code and data against the rich OS. More details can be found in Section V-A.

E. Memory Protection Outside the Execution Environment

In CaSE, processor cache is used to create a SoC-bound execution environment. As long as sensitive data resides within this environment, it will remain protected. However, SoC-bound memory, such as cache, is often small in size. When the protected application in CaSE attempts to relinquish resource for other applications, the program context needs to be saved to external DRAM. In order to protect the confidentiality and integrity of these sensitive data, any data leaving the SoC boundary needs to have a checksum, which is then encrypted along with the data. When this data is loaded back in the SoC environment, it is decrypted within the SoC and the integrity is verified with the checksum.

Due to the lack of hardware support automatic encryption/decryption like Intel SGX [20], the cryptographic protection for memory has to be provided by CaSE.

V. Design and Implementation

In this section, our design and implementation of the CaSE architecture on the i.MX53 platform is presented. Two execution modes using two TrustZone worlds are first presented. The challenges and our solutions in creating a cache-bound execution environment are then discussed. Next, details on how to isolate the running environment from the rich OS and secure the data outside the SoC are presented. Lastly, the locked cache layout in our prototype and two secure application prototypes of a cryptographic library and a kernel integrity checker in CaSE are discussed.

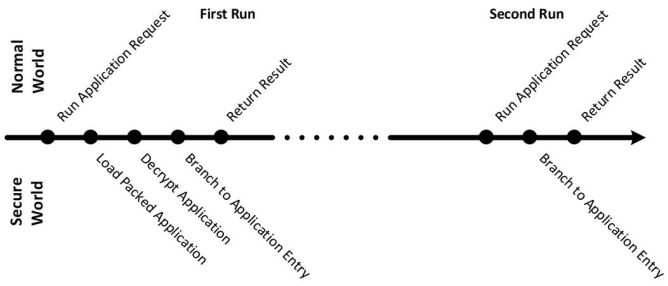

Figure 3: Execution Flow using Secure Cache

A. Two SoC-bound Execution Modes

SoC-bound execution can be performed in either the secure world or the normal world, and these two execution modes offer a trade-off between system security and performance. The overall execution flows of the two modes are introduced in the following.

1) Execution Flow Using Secure Cache: The CaSE secure mode uses secure cache to create the SoC-bound execution environment. As shown in Figure. 3, when a request to run a secure application is received, the CaSE controller loads the encrypted application in the secure cache. After being decrypted completely within the secure cache, the application will run in the secure world until it finishes and sends the results to the normal world. Since the rich OS cannot access secure cache, it is not necessary to clean the application execution environment. Figure. 3 shows that in the second run of the same application, the processor can simply branch to the application entry address in the secure cache. Thus, for frequently invoked applications such as cryptographic modules, this property can improve the system performance by eliminating repeated loading and decryption of the application. However, since we run the secure application in the secure world, it will increase the size of the code running in the secure world of the system.

2) Execution Flow Using Normal Cache: Since the normal cache can be read, flushed or invalidated by the rich OS, it seems difficult, if not impossible, to protect normal cache from a compromised rich OS. CaSE solves this problem by relying on temporal separation rather than space separation of the resource. To achieve the temporal isolation between the secure application and the rich OS, we suspend the rich OS when the secure application is running in the normal world.

As shown in Figure. 4, when a secure application needs to run, the rich OS will help load the encrypted application into the cache and set up the execution environment in the normal world. After the system switches to the secure world, the rich OS will be suspended. Then the CaSE controller will

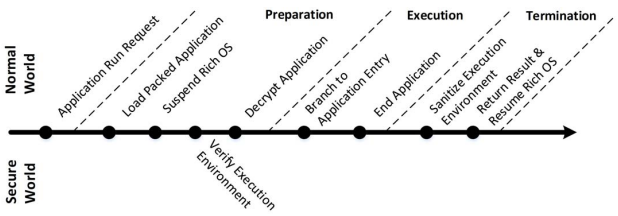

Figure 4: Execution Flow using Normal Cache

check the integrity of the application code and its execution environment. If successfully verified, the application payload is decrypted in the cache by an unpacker provided by the CaSE framework. Lastly, control flow will be directed to the application entry function from the unpacker.

For most secure applications such as kernel integrity checking, there is no information that needs to be retained between consecutive executions. However, there are some applications whose states between executions should be kept. Since the application context stored in the CPU cache cannot be protected once the control flow is directed back to the rich OS, we need to encrypt the application context before saving it to the memory.

The benefit of using normal cache is to reduce code running in the secure world since the application runs in the normal world and it cannot compromise the secure world even if it has a vulnerability. When the application execution makes use of the normal world cache, it is necessary to clean the application context including the cache lines and registers before exiting the application and resuming the rich OS.

As a result, the environment needs to be instantiated and torn down each time the same secure application runs, which has an impact on the system performance.

B. Cache-Assisted SoC-bound Execution on ARM Processor

We tackle three challenges in creating a cache-based SoC-bound execution. First, to optimize the use of the limited cache size, the mapping from memory address to cache lines has to be explored. Unfortunately, neither the architecture document nor the processor document provides details on this translation. We design experiments on real ARM processors to figure out the detailed mappings. The second challenge is to load and lock both code and data of the application along with the execution environment within the physical boundary of the processor. The third challenge is to handle self-modifying programs, particularly, the decryption of the packed application code in the cache.

1) Reversing Cache Structures: One of the key enablers of our system is the ability to reliably maintain contents in the processor cache. In order to maximize cache utilization, we need to know exactly how the cache controller maps memory addresses to cache set indexes. Only with such knowledge can we precisely and reliably utilize the entire cache. To figure out the mapping from physical addresses to cache set numbers, we first flush both L1 and L2. An *LDR* instruction is used to trigger a cache line fill on the memory location. Once the cache line is filled, we use *STR* instruction to change the values in the cache. Individual cache lines are then invalidated by set and way iteratively. When the set and way being invalidated is the same as the set and way that was used for the cache line fill, the loaded value after invalidation would be different from the value before the invalidation. By repeating this test on different physical memory addresses, we successfully reverse the cache indexing scheme. On ARM Cortex-A8 processors, we conclude that the mapping from memory to cache is linear.

2) Loading in and Locking down Cache: It is critical in CaSE to load and store the application and its execution environment completely only within the cache. ARM architecture offers the ability to lock down cache entries so that system developers can optimize the cache performance on embedded devices. We utilize this hardware function to lock all the cache lines used by the secure application. The pseudocode for cache locking is shown in Listing 1.

The first step of loading memory into cache is to configure the memory address to be cacheable. Memory in the ARM architecture can be categorized into three types, *strongly ordered*, *device*, and *normal*. Furthermore, for normal memory, there are three caching strategy, *write-back*, *write-through*, and *write-allocate*. Write-back and write-through are mutually exclusive. The caching attributes on ARM processors are controlled by various registers, including *system control (SysCtrl) register*, *aux control register*, *L2 lockdown register*, *L2 aux control register*, as well as *page table entry*. The paging table entry controls the caching strategy of the address location using the *type extension (TEX)*, *bufferable (B)*, and *cacheable (C)* bits. The combination of the 4 bits yields various caching strategies for the memory address location. However, it can be remapped via the *tex remap enable (TRE)* remapping capability in ARM. TRE allows an operating system to have finer granularity control of the types and provides additional room to store OS specific information. When TRE is enabled, memory attributes are mapped to *primary region remap register (PRRR)* and *normal region remap register (NRRR)*.

Since any data written to write-through cacheable memory is directly forwarded to DRAM, we set the cache strategy of the targeted memory area to be write-back, so that memory modifications will be buffered in the cache.

ARM processors provide hardware-assisted cache locking on L2 cache as part of coprocessor functions in CP15. The cache lock register allows system designers to enable and disable the allocation of individual cache way. Once the cache allocation is disabled, the locked cache lines will never be evicted. On the i.MX53, the granularity of L2 cache locking is by individual ways of the cache. Other platforms have

```
disable_local_irq();
enableCaching(memArea);
disableCaching(loaderCode);
disableCaching(loaderStack);
invalidate_cache(virtual address of memArea);
unlockWay(wayToFill);
lockWay(allWay XOR wayToFill);
while(has more to load in memArea)
      LDR r0, [memArea + i];
lockWay(wayToFill);
unlockWay(allWay XOR wayToFill);
```

Listing 1: Lock Memory in Cache

different cache locking functionality enabled. For instance, Tegra 3 supports a finer cache locking granularity on each cache line [35].

Using the hardware cache lock is straightforward, but the challenge lies in how to place memory contents in L2 cache instead of L1 cache. This is because we can only lock cache lines in L2 cache. The presence of a cache line in one cache level does not necessarily guarantee its appearance in the other levels. For inclusive cache, any cache line in L1 cache is also in L2 cache. Intel processors largely adopt this inclusive cache paradigm [36]. On the other hand, for exclusive cache, any line in L1 cache is not in L2 cache, and AMD processors usually follow this exclusive cache paradigm [37]. However, for ARM Cortex-A8 processors, there is no indication of inclusiveness or exclusiveness in any document. A closer examination of the cache fill strategy reveals that when there is a cache miss on data or instruction, a cache line fill to both L1 and L2 from the advanced extensible interface (AXI) bus will be triggered. Thus, cache line should be cleaned before executing *LDR* instruction on a memory address.

The code and data that are used in the cache loading have to be configured to the non-cacheable attribute. Otherwise the code itself can be loaded in the cache, causing unintended evictions of the cache lines that need to be locked. Upon completion of cache filling, the cache way can then be locked. With the cache lines locked in L2 cache, the physical memory addresses corresponding to these cache lines can be used as cache-based memory. In the ARM Cortex-A8 processor, L2 cache locking is achieved via the *L2 cache lockdown register* [38].

3) CPU Bound Application Decryption: One of the system design goals of CaSE is to offer code confidentiality, which is a challenging task. First, the file storage is in the normal world, so the application has to be encrypted while it is saved in the file system. Second, memory contents of neither the normal world nor the secure world are protected from cold boot attacks, so the application has to be encrypted in the DRAM as well. Lastly, the rich OS can be compromised by malware. As a result, the application can only be decrypted either when the rich OS is not running or in the secure world where the rich OS cannot interfere. One of the well-established binary manipulation techniques, code encryption [39], [40], is used to tackle this challenge.

We develop a cache-only packer, *CaSE Packer*, which uses AES to encrypt the code and data of the CaSE application. The entire code to be executed is loaded into cache. The unpacker then decrypts the encrypted code and places them back in the same position inside the *cached* memory, so that the code will remain in cache. CaSE packer, however, has a unique difference from existing application packers on handling the decryption process, due to the cache coherency problems in the ARM platform.

In modern processors [41], [37], [36], [38], the L1 cache (also known as the primary cache) is often split into two parts of equal size, the instruction cache (I-Cache) to speed up executable instruction fetch and data cache (D-Cache) to speed up data fetch and store. The instruction cache is often preloaded with binaries of the executable speculatively using algorithms in the hardware. However, this magic speedup falls apart when application modifies its own code in memory. In many processor architectures including ARM, the I-Cache and D-Cache are not guaranteed to be coherent, and it is up to the system software to handle cache coherency.

During the decryption of the application payload, the ciphertext needs to be loaded into the cache first. This will trigger a cache line fill into the L1 cache. When the ciphertext is decrypted, the results are stored using an *STR* instruction. Due to the close locality between the ciphertext and plaintext, the cache lines that were filled with the encrypted text will be used by the processor to store the decrypted text. Since I-Cache and D-Cache are separated in L1, the *STR* instruction will place the decrypted plaintext code in the L1 D-Cache of the processor. When the control flow is branched to the newly decrypted code, the L1 I-Cache will fetch the instructions from the L2 unified cache instead of the L1 D-Cache because of the cache hierarchy. Therefore, the processor will execute the encrypted code, which will most likely generate an undefined instruction exception.

This problem is often described as the cache coherency issue between I-Cache and D-Cache. The recommended approach to this problem is to flush out the affected portion or the entire cache to the point of coherence (PoC). Cache contents are written to memory to make sure all masters in the system see the same copy of memory content. Unfortunately, this approach is not suitable for SoC-bound execution, since flushing out contents to memory defeats the purpose of SoC-bound execution. On some platforms, it is also possible to use cache maintenance instruction *clean to point of unification (PoU)* to synchronize the internal caches. PoU is the point by which the instruction and data caches and the translation table walks of that processor are guaranteed to see the same copy of a memory location [41]. The location of PoU is platform and environment dependent.

For example, the PoU for Cortex-A15 can be either the L1 data cache or the external memory [42]. Though the location of PoU for ARM Cortex-A8 processor is not explicitly documented in the manual, we find through observations in the platform that it is possible to use the clean to PoU instruction to synchronize the internal I-Cache and D-Cache.

Besides using specific instruction, we also devise a method to manually load code into unified cache so that the same method can be used on other platforms where synchronized to PoU instruction is not suitable. Since L2 is a unified cache, synchronization between I-Cache and D-Cache is not needed. We first invalidate the cache lines in L1 for the code memory location. After L1 is cleaned, we use *write-alloc* feature in L2 cache to write to only the unified L2 cache. When write-alloc in the L2 auxiliary control is set, *STR* instruction will trigger only the L2 cache fill. We use this method in CaSE packer to enable the decryption of the application.

C. Securing Cache-Assisted SoC-bound Execution

It is not an easy task to secure the cache-assisted SoC-bound execution on ARM processors. In the following, some key design efforts to secure the execution environment are presented.

1) Key Management: Applications are encrypted and packed with a secret key to protect code confidentiality. This key should be stored in the secure storage provided by TrustZone. In our implementation, we make use of the second generation Security Controller (SCC) equipped on the i.MX53 SoC. Using the platform key that is stored in e-Fuse based secure storage on SCC, we encrypt the CaSE master key and store it along with the TrustZone code. When the system boots, the master key is decrypted and stored in the secure cache.

2) Secure Code Loading in Normal Cache: In order to load the CaSE application in the normal cache, the loading operation should be carried out in the normal world. Since the rich OS may be compromised, we must verify the integrity of the encrypted application code in the secure world. However, according to the TrustZone cache design, the secure world cannot access the contents in the normal cache. Therefore, it becomes a challenge for the integrity checker in the secure world to verify the integrity of CaSE application in the normal cache.

We use the cache array access feature in the CP15 coprocessor to overcome this difficulty. The cache array function allows a process running in the secure world to retrieve the contents of cache lines whether they are tagged as secure or non-secure. Specifically, we use the *c9* function to read the cache tags and lines into the general purpose registers for inspection. However, since the parameter used in this function is the physical cache array index instead of the cache line index and there is no one-to-one mapping from the array index to the memory location, we have to carry out several experiments to work out the mapping from the physical array index to the cache set and way number. After obtaining the cache tag, we can reconstruct the physical address for each cache line. Then, the memory contents can be used to verify the integrity of the code in the normal cache.

The application stored in the cache is indexed and tagged with physical address, while the processor executes instructions using virtual address. If an attacker inserts a malicious translation from the virtual address to the physical address, she could redirect the control flow of CaSE into any arbitrary physical address [43]. To defend against this memory address translation redirection attack, the translation needs to be locked down in the TLB cache as well.

TLB lockdown function in Cortex-A8 processor is based on the modification of eviction policy [41]. To lock down an entry, the TLB cache for the address has to be cleared out first. The TLB lockdown register is then modified to indicate which TLB line to fill for the next result of translation table walk. To fill the intended address translation at this position, a TLB preload instruction is executed to force hardware to perform a page table walk. Once the cache is filled, the TLB lockdown register is modified again to never evict the entry to achieve the TLB lock. However, similar to the L2 cache, TLB cache is also extended with an extra *NS* bit for the TrustZone architecture. Therefore, if the TLB preload instruction is performed in the secure world, the corresponding translation is for the secure world *only*. To resolve this problem, we fill the TLB in the normal world and then use the TLB data access array function in CP15 coprocessor to verify the translation.

3) Application Context Sanitization: The CaSE application context consists of the decrypted code, the decrypted data, stack, and heap. All of them are considered sensitive. When the application finishes execution, the context needs to be sanitized. There are two ways to perform the sanitization: overwriting the cache contents or invalidating the cache lines.

When cache overwriting is used, all sensitive cached memory locations are written with a known pattern using *STR* instruction. Then all the cache lines are flushed out to the DRAM such that the changes to the memory is written to DRAM and is no longer cached. When the cache invalidation method is used, all sensitive cache memory addresses are invalidated using the cache maintenance operation *invalidate by modified virtual address (MVA)*. When the cache memory is invalidated, the cache line is marked as invalid. Thus, all values in the cache line pertaining to the memory address become invalid.

We choose to use the cache invalidation method because it can be used to verify that no sensitive context information is leaked to the memory. To check for cache leaking after the execution of a protected application, we first write predefined pattern to the memory location that would be used

for the application runtime environment before loading the application. The cache is then flushed to make sure the predefined pattern is in the DRAM. At the end of execution, cache lines used for application execution are invalidated. If there is no cache leak, the result of the *LDR* instruction should return the pre-defined pattern. Otherwise, if the sensitive cache lines had been evicted during the execution of the protected application, then the value will be different.

4) Handling Cache Coherency between TrustZone Worlds: The cache coherency issue between the normal world and the secure world creates not only the challenge for integrity verification, but also the delivery of computation output. In some application configurations, when the output is cached in the secure world, it is not immediately available to the normal world. For our cryptography library prototypes, the results of encryption that are cached in the secure world are not accessible by the CaSE driver in the normal world until the cache lines are evicted.

To make the output immediately available to users in the normal world, the CaSE application in the secure world has to flush the outputs that are being cached. However, we cannot simply flush the entire cache, since sensitive contents in the cache will also be written to DRAM. There are two methods to solve this problem: clean by MVA or clean by set and way of individual level of cache. When the clean by MVA method is used, CaSE needs to invoke clean by MVA for all the memory addresses of the output buffer. When the clean by set and way method is used, CaSE needs to walk through all the non-sensitive sets in all ways across all cache layers. More specifically, since L1 does not provide locking capability, there is no way to know if a line contains sensitive data or the computation results. Therefore, all the lines in L1 are clean. However, for L2, we know the memory organization of cache ways that are locked. Thus, we can clean all the non-sensitive sets and ways. Our implementation uses the aforementioned method based on the size of the output. When the size of the output is large, it is better to flush the all the non-sensitive cache.

On the i.MX53, *c7 c10* system coprocessor function is invoked with opcode 1 to clean the set and way, and the same function is invoked with opcode 2 to clean by MVA to point of coherency (PoC). Point of coherency is where the processor core and other masters such as DMA controller see the same copy. For i.MX53, PoC is the DDR memory.

5) Securing Across Power States: An energy-conscious mobile device will switch the processor into different power states to save energy. When the processor is put in the sleep state, power supply to processor cache is cut down, so all data stored in the cache will be erased. This poses a challenge for CaSE, which uses processor cache to create the execution environment.

A simple solution is to keep the cache powered. Both L1 cache and L2 cache can be placed in a different power domain than the integral core. However, this approach has its drawback in power consumption. Modern cache is often constructed with SRAM, which consumes more power than DRAM. An alternative method is to store the cache context in memory that is physically inside the SoC, such as the on-chip RAM (OCRAM). However, many BSPs have claimed the usage of OCRAM for other subsystems [44]. Furthermore, some platforms might not have built-in support to include OCRAM in the secure domain.

In CaSE, we adopt the method that encrypts the cache and saves it in DRAM when the device is in power saving modes. When the system sleeps, the rich OS notifies the CaSE controller to encrypt the cache contents with the master key and then save them into the DRAM. When the system resumes, the contents are loaded back from the DRAM into the cache along with the master key recovered from the secure storage.

D. Application Development

CPU Cache is one of the key elements to improve system performance in modern processor design. Cache-assisted SoC-bound execution system will inevitably have an impact on the system performance when locking down portions of cache for special usage. Therefore, it is important to optimize the usage of locked cache. We first present the general layout of the locked cache and then our prototypes of two secure applications using CaSE.

1) Layout of Locked Cache Way: In a typical layout of CaSE application, we place the master key, which is used to decrypt CaSE applications in the first set of the way, followed by the encrypted code and data sections of the CaSE application. CaSE packer code and the environment setup code are immediately after that. Lastly, the rest of the cache lines in the way are used for stack and heap of the application.

There are several essential components for the execution of a binary image, including the libraries, the virtual memory address space layout, code packer/unpacker, and the stack and heap. First, applications cannot use the library provided by the rich OS, since the library integrity cannot be guaranteed. Therefore, CaSE applications need to be statically linked into the binary itself. For our prototypes, we make an effort to modify all the code so that they are self-contained. This is also a byproduct of the effort to minimize the binary code size.

Second, the physical address of the application address space needs to be carefully crafted to fit in a single cache way without causing a collision on the same cache set. We create a set of linker scripts to work with our customized CaSE loader instead of using the default loader and linker script. More specifically, the linker script configures the start address of the binary and the section arrangements.

Third, a CaSE packer is used to encrypt the application binary to provide code confidentiality outside the cache. The packer in our prototype uses AES encryption in CBC mode.

Using our own loader simplifies the design of the packer and unpacker, since it is not necessary to implement all the ELF file standards.

Lastly, a custom heap and stack is provided. The stack is allocated by the CaSE loader from the cache memory. Furthermore, we create a simple heap management library, which allocates heap spaces in the cached memory, as well.

2) Two Secure Applications: Unlike the previous approaches [22], [24], [26] that are designed for a specific algorithm, CaSE offers a generic execution environment. In other words, users do not need in-depth knowledge of the application to create a SoC-bound execution. We develop two secure applications using CaSE application framework.

First, we build a cryptography library by porting AES, RSA, and SHA1 from polarSSL library [45]. Cryptography is one of the fundamental building blocks in modern day computer and network security. Due to the small size of cryptography libraries, it is feasible to place them in the secure cache. The unique advantage of executing in the secure world is the ability to switch context without environment sanitization. As shown later in the experiments, this execution mode has little performance impact on the rich OS, yet offering enhanced security protection. We place AES, RSA and SHA1 all into one library called CaSE crypto library. By combining SHA1 and RSA in the same library, we are able to save some code space due to the use of shared library. Lastly, we need less than one L2 cache way to construct the CaSE cryptography library execution environment.

Second, we build a kernel integrity checker that is invoked periodically to verify the integrity of the rich OS kernel code page. In particular, we calculate a SHA1 checksum of the all the kernel code pages to make sure that it is not modified by any malicious software. Most development efforts are to remove the dependency on the rest of the polarSSL library and the c standard library. We run the kernel integrity checker as normal world SoC-bound execution application, since there could be different implementations of system integrity check and it is difficult to include all variances in the secure code base of the system. Thus, the normal world execution environment is more suitable for the kernel integrity checker.

VI. Experimental Evaluation

In this section, we first introduce the experiment setup in VI-A. The sizes of various system codes are examined in VI-B. Cache behavior on the platform is studied in VI-C. The last part of the evaluation in VI-D examines the performance of CaSE application system as well as the performance impact of cache locking on the system.

A. Experiment Setup

We implement our prototype of CaSE on the FreeScale i.MX53 mobile development board. It features a single ARM Cortex-A8 processor with 1GB DDR3 DRAM and 128 KB onboard internal RAM (iRAM) and 16 KB secure iRAM. The system boots with onboard flash along with the uboot and kernel supplied by the Micro-SD card inserted. We use the FreeScale Android 2.3.4 platform with a 2.6.33 Linux kernel. There are two levels of cache in the ARM Cortex-A8 processor. Both the L1 data cache and L1 instruction cache are 4 way 128 set associative cache with 32 KB size. L2 cache is an 8 way 512 set associative cache with 256 KB size. The Android OS is ported from secure domain to the normal domain based on the Board Support Package (BSP) published by Adeneo Embedded [44].

B. Code Size

The system TCB consists of three components. The first component is the trusted boot code, which is about 500 source line of code (SLOC). The second component is CaSE controller, which is responsible for handling CaSE environment initialization and clean up. It has approximately 500 SLOC of code. The third component is related to specific application implementation. In CaSE, we use SHA1 to check the integrity of isolated execution environment cache, and AES to encrypt application state while it is paused. The SHA1 implementation is 166 SLOC, and the AES is 579 SLOC. In total, there is 745 SLOC for the cryptographic libraries. This additional SLOC does not necessarily have to be included in the TCB if the system only requires secure execution mode.

While SLOC number offers a good estimation of the size of the TCB, it is also important to show the size of the binary code for SoC-bound execution. This gives an idea of the feasibility of fitting the application in the cache. On the Cortex-A8 processor, one L2 cache way is 32 KB.

Application	Code+Data (KB)
AES	2.4
RSA	10
SHA1	5
CaSE Crypto Lib	17.4
Kernel Integrity Checker	6.6
CaSE Packer	2.8
Packed CaSE Crypto Lib	20.4
Packed Kernel Checker	9.5

Table I: CaSE Application Size

The code size shown in Table I is compiled from C and assembly source code using ARM Thumb-II encoding. We turn on the ARM interworking mode during code generation in the compiler. This increases the code size but makes it easier to interwork with the ARM code in our security monitor. With careful coding between function calls, one can remove this compiler flag to further reduce the size of the binary code.

C. Cache Bound Verification

Cache-bound verification is designed to verify the security attributes of CaSE. Specifically, there are several aspects of the system we want to verify. First, we need to verify that the location of the CaSE application data indeed exists only in cache, and there is no cache leak during the execution of the program. Second, we want to verify that the processor cache that has been locked down using hardware functions cannot be read or written by the DMA attacks. Third, we examine the interaction of the locked cache lines with cache maintenance operations.

1) Verifying Applications Exist Only in Cache: The primary goal of this test is to show that the sensitive application is indeed in and only in the processor cache. Furthermore, we want to verify that the CaSE execution environment will not leak any application code or data into the DRAM at any point in time during the execution. Without hardware support, it is difficult, if not impossible, to verify this property for each processor clock cycle the application uses. We choose two points in the execution flow that are likely to show leaked contents if the cache had been flushed to memory. The first point of inspection is at the completion of unpacking action. The second point is when CaSE execution completes, but before the environment is cleaned up.

In this test, we use the packed kernel check application as the test case. We inspect right after unpacking and when the kernel check completes. For both tests, we instrument the code to invoke leakage check routine right after the unpack operation and kernel check. The leakage routine is stored in memory outside CaSE application. The check routine will invalidate all the cache lines occupied by the CaSE application, and then read back the memory location. If the read back value is not the default value for those memory (0xFFFFFFFF), then there is a leak from the protected application. In both points of execution, we detect no leak in the CaSE environment from cache to memory. In order to assure that there is no integer value of 0xFFFFFFFF leaked, we also try another pattern (0xABABABAB), and the results are the same.

2) Verifying the Effect of DMA Attack on Cache: One of the main design objectives of CaSE execution is to defend against compromised rich OS. Even though the rich OS is paused during the execution of CaSE application in the normal world, it is still possible for the rich OS to program an I/O device to perform DMA memory read and write to the normal world memory. To see how DMA will interact with cache lines, we program the serial controller to perform DMA read and write to the memory that we lock in cache. More specifically, we load the application in the normal world cache, and use a kernel module in the normal world to program the serial port using DMA to dump memory over serial. We observe that the dump fails to extract contents from cache.

3) Verifying the Effects of Maintenance Operations on Locked Cache: This test studies the effects of cache maintenance operations on locked cache lines on the i.MX53 and verifies that secure cache cannot be manipulated by cache maintenance operations executed in the normal world.

Common cache maintenance operations include cache clean and cache invalidation. These cache maintenance operations are coprocessor functions that can only be initiated by the CPU itself. Thus, attackers can only launch cache maintenance attack from the rich OS. When one CaSE application is running in the normal cache, the rich OS is suspended, so there is no software attack from the rich OS. However, when the rich OS resumes the system control, it can attempt to use cache maintenance operations to launch attacks on the secure cache, where the master key and the secure CaSE application are stored. In this experiment, we want to verify that malicious code loaded in the rich OS cannot clean or invalidate the secure cache.

We begin the experiment by writing 0xFF to all the memory buffers. We then fill one cache way with normal world cache lines of pattern 0xAB, and another with secure cache lines with pattern 0xBC. Once the two ways are locked, we use the CaSE driver in the normal world to execute the cache maintenance operation. To see if cache clean instruction in the normal world can evict the locked cache lines, we execute clean instruction on all the locked cache lines in both the secure world and the normal world. If the cache contents were written to memory due to the clean instruction, the value read back will be 0xAB for normal world and 0xBC for secure world. In our experiment, memory in the normal world reads back as 0xAB while the memory in the secure world reads back as 0xFF. This verifies that cache clean instruction invoked by the normal world will not be able to affect cache lines in the secure world. We follow a similar procedure for cache invalidation, except that INVD instruction is used instead. We observe that cache lines in the normal world are invalidated, because the read back value is 0xFF, while values read back from the cache lines in secure world remain 0xBC. Therefore, the rich OS cannot use cache maintenance instruction to manipulate the cache lines of the secure world.

D. SoC-bound Execution Performance

We study the performance of the system by examining the time breakdown of the CaSE application execution. We also compare the performance difference when the secure application is running in the normal cache or the secure cache.

1) CaSE Isolated Application Performance: As a case study, we use the kernel integrity checker as a normal world application. The packed CaSE kernel integrity check application is 9.5 KB in size. The timing breakdown for each major operation in the CaSE execution is shown in Table II.

Operation	Time (μs)
Environment Preparation	613
Environment Integrity Check	1540
CaSE Unpacking	5973
Kernel Check	18676
Environment Cleanup	412
Total Time	27214

Table II: Kernel Integrity Checker in Normal Cache

From the time breakdown, we can see that though environment setup and cleanup consume some processor cycles, the major computation overhead originates from the unpacking process, which decrypts the encrypted CaSE application payload. The entire kernel check takes 0.02 second to complete, and the application context saving time is 94 μs.

2) CaSE Secure Application Performance: Using the crypto library as a case study for the CaSE secure execution mode, we measure the benchmarks for a secure cache execution similar to the normal cache execution. Table III shows the time breakdown of a secure call to perform encryption using AES CBC mode. In the secure mode, the cache is protected against the compromised rich OS. Therefore, it is not necessary to clean up the execution environment.

Operation	Time (μs)
World Switching	2.6
AES CBC Encrypt (1KB)	443
Output Synchronization (1KB)	2
Total Time	447.6

Table III: AES Encryption in Secure Cache

3) Performance Trade-off between Execution Modes: To find out the impact of SoC-bound execution environment on application performance, we run AES, RSA, and SHA1 in different environments and compare their performance. First, we port the application into a kernel module and load the module into the rich OS to measure the performance without any security enhancement. Second, we run the application in the two CaSE execution environments, one in the normal world and the other in the secure world. We consider that the first experiment should achieve similar performance as other kernel encryption solutions, and should serve as a good baseline for comparison. On the other hand, the CaSE execution will suffer performance penalty for the enhanced security.

The experimental results on AES algorithm are shown in Figure. 5. The performance of secure executed AES is almost identical to that of generic AES. The secure AES has a small advantage over the generic kernel AES when the memory buffer to be encrypted is small. This is due to preloaded cache lines for the AES data section. For

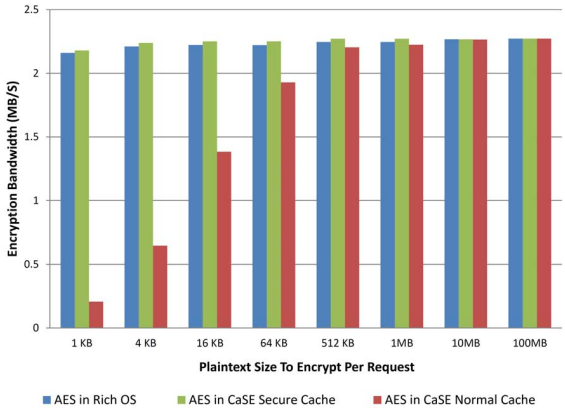

Figure 5: AES Speed Comparison

smaller size encryption requests, the normal cache execution is significantly slower than the other two methods. This is because the environment is created and destroyed for each request in order to protect the confidentiality and integrity of the execution environment. However, as the size of the plaintext increases, the difference in the encryption bandwidth diminishes. This is because the overhead to create and destroy the environment becomes insignificant.

We have also performed the same set of experiments on RSA algorithm and SHA1 algorithm. The results for RSA algorithm are shown in Figure. 6. In this experiment, we measure the number of 1024-bit RSA decryptions that the system can carry out in one second. Similar to AES, the normal cache execution takes a penalty in the environment initialization and clean up. However, as the number of messages in the request becomes larger, this fixed cost can be ignored. Lastly, we also benchmark the performance of SHA1. We build up our test case by sending fixed size 512 byte packet to the SHA1 module to calculate the hash. Due to simplicity of SHA1, the normal world execution overhead is high when the number of messages per request is low. Similar to RSA and AES, the environment penalty becomes small as the number of messages increases.

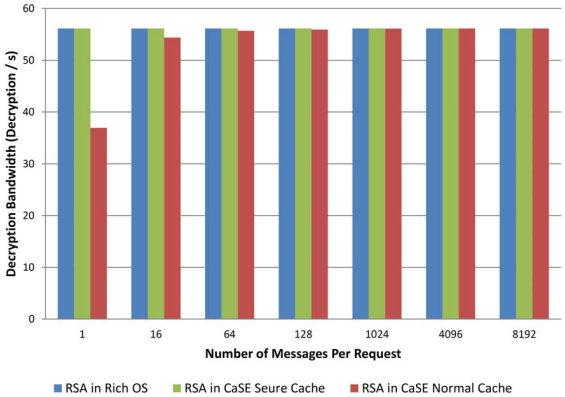

Figure 6: Comparison of RSA Operation

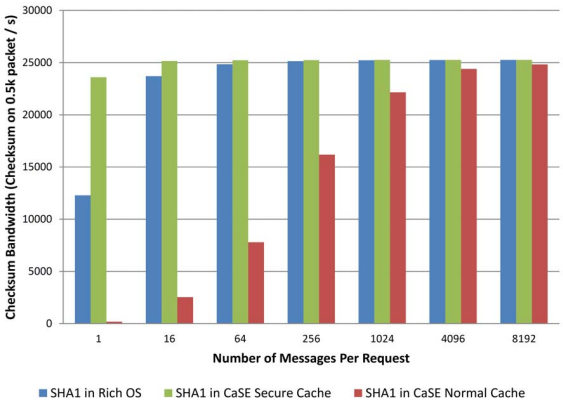

Figure 7: Comparison of SHA1 Operation

4) Impact of Cache Locking: Cache is originally designed to enhance system performance. When it is locked intentionally for non-performance reason, the system suffers. We use various benchmarking tools to assess the impact of cache locking on the system performance when different portions of cache are locked down. In our implementation, only one way of the L2 cache is locked to reduce the impact on the system.

This experiment is designed to explore the trade-offs between the size of the CaSE application and the impact on system performance due to its monopoly on the L2 cache of the system. We use three benchmarking tools, randspeed [46], linpack [47] and AnTuTu [48]. RandMem measures the performance of random access on large array of memory [46]. The performance benchmark of this tool relates closely to the performance of the memory subsystem. Therefore, with more cache locked away, the system suffers bigger penalty in memory performance. Since Linpack measures integer operation speed of the system [47], the reduction in L2 cache has a smaller impact in LinPack benchmark. Lastly, AnTuTu [48] is a comprehensive benchmark suite. It measures the performance of the system in integer computation, float point operation, 2D and 3D graphic rendering etc. AnTuTu can provide the overall system impact when the L2 cache is locked.

As shown in Figure. 8, locking one out of eight ways in L2 cache has at most 3% performance penalty. However, the overall system performance degrades more quickly when more than 60% of the L2 cache is locked and becomes unavailable. This pattern is consistent with other benchmarks in the AnTuTu suite including 2D GPU, single thread integer operation, and multi-thread integer operation.

VII. Security Analysis

With the physical possession of the mobile device, adversaries have two attack vectors, software attack and cold boot attack. We assume that attackers with physical access can only examine contents of the physical memory but not the cache and registers inside the processor. The attack model is

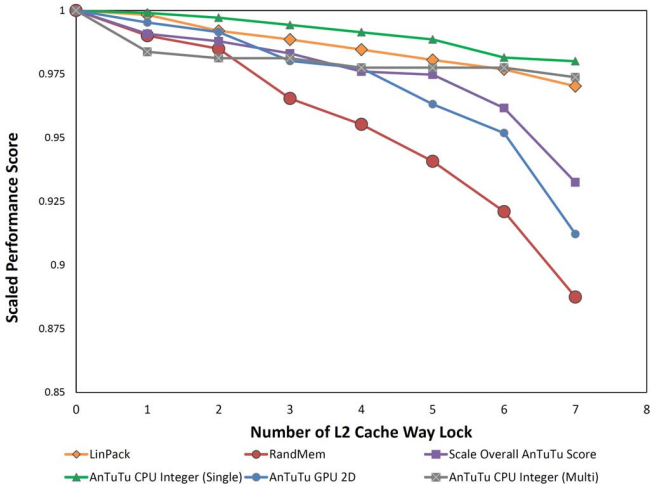

Figure 8: Performance Impact of L2 Cache Locking

summarized in Table IV. The adversary who compromises the rich OS can gain unrestricted read and write access to the normal memory and cache, but neither the secure memory nor the secure cache. The adversary who utilizes cold boot attacks can gain unrestricted read access to the memory of both the normal world and the secure world, but not the cache contents.

CaSE is designed to protect confidentiality and integrity of the application. Applications are encrypted with a checksum while stored in memory. Code and the data of the application are only decrypted in the cache-assisted SoC-bound execution environment.

A. Software Attacks from Compromised Rich OS

In the CaSE execution using normal cache, the rich OS is suspended by disabling all local interrupts. There is no Non-Maskable interrupt (NMI) on the i.MX53. When CaSE is built on platforms with NMI, system designers should redirect these interrupts to TrustZone temporarily for secure processing. With the OS suspended, it cannot launch any attacks to compromise either the integrity or the confidentiality of the processor cache.

In the CaSE execution using secure cache, the rich OS cannot read or modify the secure memory or the secure cache. This is because the memory space is completely separated between the two worlds by TrustZone. To improve performance during context switching, the CaSE execution environment in the secure cache is not sanitized. Because of this design choice, the rich OS may also attempt to use cache maintenance instruction to evict the secure cache out to DRAM, and then use cold boot attack to read out the DRAM contents. However, we verify via experiments that cache maintenance instructions executed in the normal world affect only the normal cache. This is because secure cache is handled differently by the cache controller due to the

Attack Vector	S Mem		NS Mem		S Cache		NS Cache	
	Rd	Wr	Rd	Wr	Rd	Wr	Rd	Wr
Software Attack			✓	✓			✓	✓
Cold Boot Attack	✓		✓					

Table IV: Attacker Capability on ARM TrustZone

TrustZone protection.

The rich OS can also launch an impersonation attack. The compromised rich OS can send an application request as the original user. CaSE does not have any built-in mechanism to mitigate this attack. However, an application can use mutual authentication to thaw this attack. For the cryptographic modules in our prototype, adversaries can launch chosen plaintext attack and chosen cipher text attack. However, most modern cryptography methods, including AES, are designed to resist such attacks.

DMA requests on secure memory from peripheral devices, such as LCD controller, are prevented by the TrustZone-aware DMA controller (DMAC). Therefore, the rich OS cannot program peripheral devices in the normal world to read or write secure memory.

Lastly, the compromised rich OS can change the power state without notifying the CaSE controller in the secure world. In this case, all cached sensitive contents of the CaSE controller will be lost. On subsequent invocation to CaSE controller, all requests will be dropped. We do not consider this as a real threat, since the attacker who already has physical access can simply power off the device to deny services.

B. Unrestricted Memory Read from Cold Boot Attack

Cold boot attacks are capable of reading both the normal world memory and the secure world memory. Since a cold boot attack physically removes DRAM chip from the system, we assume it will be too difficult for the attacker to modify the value in DRAM circuit without interrupting the operation of the system. Therefore, only the confidentiality of the memory is compromised, but not the integrity. In CaSE, application contexts and application binaries are always encrypted while in DRAM. The key for the encryption is stored in the processor cache or in the on-chip secure storage while the system is in power saving mode. Thus, it is protected from cold boot attacks.

Since modern processor cache is built using SRAM which does exhibit the remanence effect similar to DRAM, the compromised OS can attempt to reboot the system to run on a malicious OS to exploit this fact to extract sensitive information from the cache. However, the malicious OS would fail the high assurance booting process. Furthermore, the SoC firmware on the ARM Cortex-A8 processor resets the cache contents upon power reset event. Therefore, we can prevent this attack too.

VIII. Discussion and Future Work

A. Migrating CaSE to Other Platforms

Though our prototype implementation of CaSE is on the i.MX53 development board, the system design is widely applicable to other hardware platforms that can provide isolated execution and SoC-bound memory storage.

1) Multi-core Processors: The i.MX53 has a single core processor. For multi-core processors, it is no longer necessary to suspend the execution of the rich OS. When the SoC-bound execution runs on a subset of the cores, the other cores can continue executing the workloads of the rich OS. It will bring some new challenges. Though applications running in the secure cache can still be protected by the TrustZone isolation, applications running in the normal cache may be compromised by the rich OS running on the other cores. Thus, system designers need to rely on some dedicated system features in the multi-core SoC to enable the cache isolation. For instance, new cache controllers in both ARM and AMD platforms [56], [57] have the capability to assign individual last level cache block to specific processor cores.

2) On-chip Memory: To improve system performance on embedded devices, SoC designers are continuously increasing the use of silicon layout for allocating more on-chip memory. The average percentage of layout used for memory is 80% in the year 2008 and has been rising [58]. There are three main categories of on-chip memory, SRAM, DRAM, and ROM, where SRAM and DRAM are more suitable to construct SoC-bound execution.

In our implementation of CaSE on the i.MX53, processor cache (SRAM) is used as memory space for SoC-bound execution. There are other choices of on-chip memory as well. For example, the internal DRAM (iRAM) has been used to store cryptographic data of AES [18]. The size of iRAM varies in different systems, and it is usually small. For example, there is only 144 KB iRAM for i.MX53 [59] and 272 KB iRAM for i.MX6 [60]. Furthermore, certain Board Support Package (BSP) uses the iRAM for other purposes such as video processing [44]. Another important difference between iRAM and processor cache is the isolation mechanism. Unlike cache, iRAM occupies a range of physical address space. The protection of this address space from the compromised rich OS or malicious DMA-capable I/O devices might not be available, and it is platform specific [59], [60].

Another important consideration is the remanence effect for on-chip memory storage. On-chip memory is free from the physical memory extraction in cold boot attacks due to its proximity on die. However, they still exhibit remanence effect just as the external DRAM. When the contents are not sanitized upon system reset, sensitive information could leak out during the boot up process. System designers should verify that the on-chip memory is well protected across different power state changes including full system reset.

	Platform		Software Attack				Cold Boot Attack	
	x86	ARM	Data Confidentiality	Data Integrity	Code Confidentiality	Code Integrity	Data Confidentiality	Code Confidentiality
Hardware-Assisted Execution								
Intel VT-x/AMD-v based [49]	✓		✓	✓		✓		
Intel TXT/AMD SVM based [50]	✓		✓	✓		✓		
Intel SGX based [20]	✓		✓	✓	✓	✓	✓	✓
System Management Mode (SMM) based [51]	✓		✓	✓		✓		
Coprocessor based [52], [53]	✓		✓	✓		✓	✓	
TrustZone based [54]		✓	✓	✓		✓		
SoC-bound Execution								
Register based [24], [22]	✓	✓					✓	
Cache-based [26], [55], [18]	✓	✓					✓	
On-Chip Memory Based [18]		✓					✓	✓
CaSE		✓	✓	✓	✓	✓	✓	✓

Table V: Comparison of Secure Execution Environment under Software Attack and Cold Boot Attack

For example, on the i.MX53, processor firmware resets the cache contents when the processor is rebooted. This can be confirmed by the *L1RSTDISABLE* and *L2RSTDISABLE* bits on the auxiliary control register of the Cortex-A8 processor.

3) Secure Information Flow: It is critical to secure the information flow between the external DRAM and the on-chip memory when adopting CaSE on other platforms. Our implementation on the i.MX53 relies on the isolation provided by TrustZone and the ability to lock memory contents in cache to prevent malicious attacks. Program contexts are encrypted before being written to the external DRAM. Though cache locking has been supported by a wide range of ARM processors, most x86 processors still do not support fine-grained cache manipulation. On those processors, information flowing out of the processor due to cache contention can be protected by temporal separation, which clears sensitive contents between the two executions of the protected application.

On some new platforms that support I/O coherent cache [61], [62], it is allowed for an I/O device to access cache contents. To protect against potential DMA attacks, system designers can use the inputoutput memory management unit (IOMMU) [36] or the system memory management unit (SMMU) [63] to secure DMA operations.

B. Supporting Unmodified Applications

CaSE has made it possible to execute arbitrary self-contained applications in a secure execution environment that provides both confidentiality and integrity for the code and data of the application. To support non-trivial unmodified legacy applications, we must enhance the platform with support for encrypted memory paging and verified system calls in the untrusted rich OS.

1) Encrypted Memory Paging: In our current implementation of CaSE, applications are loaded and decrypted completely within the cache. However, for large size applications that fail to fit in the cache, the current method of SoC-bound execution will not be sufficient. CaSE can be extended to support larger applications by keeping only the most recently used memory pages decrypted inside the SoC while leaving other pages encrypted in memory.

Due to the lack of hardware supported enclave such as Intel SGX [20], memory encryption and decryption will be triggered by the software. In order to provide seamless support for memory paging into and out of the SoC boundary, the page fault handling routine has to be interposed. When the application accesses a page that is not in the SoC, the page fault can then either be handled by the rich OS [18] or the security monitor in TrustZone [15].

While it is fairly straight forward to extend CaSE to handle applications that do not fit in the cache, similar to other memory encryption system [18], [64], [65] there is a high performance penalty for applications that frequently swap memory pages [18].

2) Verified System Calls to Rich OS: Many non-trivial applications require OS support to perform meaningful tasks. Our current implementation of CaSE can be further extended to support the use of system call. Before the system call is made, the application is paused and the application context will be encrypted and then stored in DRAM. The system call request is then forwarded to the untrusted rich OS. Upon completion of the system call, the application is resumed by decrypting the application context in the processor cache. Unfortunately, it is not sufficient to simply enable system call from the application. When the OS is compromised, it is possible for the malicious OS to launch Iago attacks [66] where the result of system calls is manipulated to subvert a protected application. Protecting unmodified applications in commodity operating system has been an active area of research [67], [68], [69]. CaSE can benefit from system call behavior verification techniques from these systems [68].

IX. Related Work

To protect the wide spread of software vulnerabilities in applications and operating systems, hardware-assisted isolation has been widely adopted in both x86 and ARM

processors [50], [51], [20], [15], [54]. On the other hand, physical memory disclosure attacks [16], [17] achieve complete memory exposure through a different attack vector. CaSE aims to provide a SoC-bound execution environment that can defend against both attacks. Our work is closely related to the research on isolated execution environments and cold boot resistant computations.

A. Isolated Execution

A line of research on isolated execution environments [50], [70], [49], [71], [72], [27], [51], [73], [74], [75], [20], [54], [4], [10], [12], [9], [8], [7], [14], [76], [53], [52] has attracted much attention as security becomes one of the most important aspects in modern information systems. One of the key challenges is to bootstrap a trusted environment and to isolate it from the untrusted environment. Earlier work focuses on bootstrapping an isolated environment using a high privileged entity, such as hypervisor [70], [49], [72] or System Management Mode (SMM) [51].

As vulnerabilities are discovered routinely in the highly complex modern OS, hardware-assisted protection is widely used in information systems due to the attractive property of shielding applications from potentially compromised OS [50], [15], [54], [51], [4], [10], [12]. Intel Trusted Execution Technology (TXT) and AMD Secure Virtual Machine (SVM) are used in [50] to create a trusted isolated execution environment for protecting security sensitive applications. Recognizing the lack of efficient context switching in the TXT technology, Intel recently proposed Intel Secure Guard Extension (SGX) [20] to provide an efficient secure enclave for isolating sensitive applications. Moreover, data stored outside of the processor bound enclave is automatically encrypted by the processor. We share the same design concept as the Intel SGX and target at achieving the same security goals. However, SGX is a processor extension on the x86 platform, while CaSE builds on the commodity TrustZone enabled ARM systems.

For mobile devices that are running on ARM processors, TrustZone has been widely adopted in [54], [4], [10], [12], [9], [8], [7], [14]. Different from the previous works that utilize TrustZone, CaSE attempts to address the threat of cold boot attacks on the system. Lastly, coprocessor has also been proposed to achieve secure computation in adverse environment [76], [53], [52]. CaSE runs on the commodity hardware and does not require additional dedicated coprocessor for code execution.

B. SoC-bound Execution

In order to defend against cold boot attack, sensitive information has to be kept in memory areas outside the DRAM. There are several types of memory that are inside the physical boundary of the SoC, namely, register, cache, and on-chip RAM. Several research works [24], [22], [23], [77] use register to store cryptographic sensitive materials. In [25], [26], [18], [55], the sensitive cryptographic materials are stored in the processor cache. Alternatively, OCRAM is used in [18]. Sentry [18] is closely related to our work. It also uses cache locking function for CPU-bound execution. However, similar to other cache-based SoC-bound execution, the security of Sentry builds on the strong assumption that the mobile OS can be trusted. We address the risk of compromised OS attack in CaSE. Lastly, even though the use of TrustZone and support for unmodified application are briefly mentioned in [18], no further description of implementations is provided.

Table. V compares CaSE with other approaches towards secure execution environment in terms of the security protections under software attacks and cold boot attacks. As indicated in the table, CaSE is the first to provide a SoC-bound execution environment on ARM platforms against both cold boot attacks and software attacks. Furthermore, both code and data of the program are protected in CaSE.

X. Conclusion

In this paper, we present CaSE, a TrustZone enabled SoC-bound execution environment to protect against both cold boot attack and compromised rich OS attack. CaSE offers two modes of operation, SoC-bound execution in the normal cache and SoC-bound execution in the secure cache, which provide a trade-off between system performance and security. We build a crypto library and a kernel integrity checker to demonstrate the practical usage of our system on real ARM platform. The experimental results show that CaSE environment only introduces little performance overhead.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by US National Science Foundation under grants CNS-1217889, CNS-1446478, CNS-1405747 and CNS-1443889. Dr. Kun Sun is supported by U.S. Office of Naval Research under grants N00014-15-1-2396 and N00014-15-1-2012. Ning Zhang is supported in part by Raytheon advanced scholar program.

References

[1] "World has 6 billion cell phone subscribers." http://www.huffingtonpost.com/2012/10/11/cell-phones-world-subscribers-six-billion_n_1957173.html.

[2] "ARM strategic report." http://ir.arm.com/phoenix.zhtml?c=197211&p=irol-reportsannual, 2014. Accessed: 2015-04-30.

[3] S. Fox, "51% of u.s. adults bank online." http://www.pewinternet.org/2013/08/07/51-of-u-s-adults-bank-online/.

[4] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, "Trustworthy execution on mobile devices: What security properties can my mobile platform give me?," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (Trust 2012)*, June 2012.

[5] "Mobile threat report: Whats on the horizon for 2016," 2015.

[6] "ARM Security Technology, Building a Secure System using TrustZone Technology," apr 2009.

[7] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun, "Smartphones as practical and secure location verication tokens for payments," in *Proceedings of the Network and Distributed System Security Symposium*, NDSS'14, 2014.

[8] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for ARM," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pp. 558–569, 2014.

[9] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure online mobile advertisement attestation using trustzone," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015, Florence, Italy, May 19-22, 2015*, pp. 75–88.

[10] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment," in *Proceedings of the Network and Distributed System Security Symposium*, NDSS'15, 2015.

[11] J. Winter, "Trusted computing building blocks for embedded linux-based arm trustzone platforms," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pp. 21–30, ACM, 2008.

[12] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 67–80, ACM, 2014.

[13] "Sierraware." http://www.sierraware.com/open-source-ARM-TrustZone.html.

[14] "Samsung knox." https://www.samsungknox.com/en.

[15] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 90–102, ACM, 2014.

[16] 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," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[17] T. Müller and M. Spreitzenbarth, "Frost," in *Applied Cryptography and Network Security*, pp. 373–388, Springer Berlin Heidelberg, 2013.

[18] 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 *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 177–189, ACM, 2015.

[19] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable hardware for high-security/high-performance embedded systems: the safes perspective," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 2, pp. 144–155, 2008.

[20] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, p. 10, 2013.

[21] A. Baumann, M. Peinado, and G. C. Hunt, "Shielding applications from an untrusted cloud with haven," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pp. 267–283.

[22] T. Müller, F. C. Freiling, and A. Dewald, "Tresor runs encryption securely outside ram," in *USENIX Security Symposium*, 2011.

[23] 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*, pp. 73–82, ACM, 2011.

[24] J. Gotzfried and T. Muller, "Armored: Cpu-bound encryption for android-driven arm devices," in *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pp. 161–168, IEEE, 2013.

[25] J. PABEL, "Frozencache mitigating cold-boot attacks for full-disk-encryption software," in *27th Chaos Communication Congress (Berlin, Germany*, 2010.

[26] L. Guan, J. L. amd Bo Luo, and J. Jing, "Copker: Computing with Private Keys without RAM.," in *In Network and Distributed System Security Symposium (NDSS)*, 2014.

[27] A. Vasudevan, J. McCune, J. Newsome, A. Perrig, and L. Van Doorn, "Carma: A hardware tamper-resistant isolated execution environment on commodity x86 platforms," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pp. 48–49, ACM, 2012.

[28] A. Aziz, "The evolution of cyber attacks and next generation threat protection," *RSA Conference*, 2013.

[29] E.-O. Blass and W. Robertson, "Tresor-hunt: attacking cpu-bound encryption," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 71–78, ACM, 2012.

[30] C. Tarnovsky, "Attacking hardware: Unsecuring [once] secure devices," 2009.

[31] E. M. Chan, J. C. Carlyle, F. M. David, R. Farivar, and R. H. Campbell, "Bootjacker: Compromising computers using forced restarts," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, 2008.

[32] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: Toward snoop-based kernel integrity monitor," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[33] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 591–604, May 2015.

[34] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 191–205, IEEE, 2013.

[35] *Technical Reference Manual - VIDIA TEGRA 3*, Sep 2013.

[36] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Sep 2013.

[37] *Advanced Micro Devices. Amd64 Architecture Programmer's Manual*, may 2013.

[38] *ARM Cortex-A8 Processor Technical Reference Manual*, June 2012.

[39] A. Griffiths, "Binary protection schemes." https://www.exploit-db.com/docs/59.pdf. Accessed: 2016-03-01.

[40] M. Oberhumer, L. Molnár, and J. F. Reiser, "Upx: Ultimate packer for executables," 2004.

[41] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*, Dec 2011.

[42] *ARM Cortex-A15 MPCore Processor, Technical Reference Manual*, june 2013.

[43] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang, "Atra: Address translation redirection attack against hardware-based external monitors," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 167–178, ACM, 2014.

[44] "Reference bsps for freescale i.mx53 quick start board." http://www.adeneo-embedded.com/Products/Board-Support-Packages/Freescale-i.MX53-QSB. Accessed: 2015-04-30.

[45] P. Bakker, "Polarssl." https://github.com/ARMmbed/mbedtls. Accessed: 2015-04-30.

[46] R. Longbottom, "Roy longbottom's pc benchmark collection," 2014.

[47] J. Dongarra and P. Luszczek, "Linpack benchmark," *Encyclopedia of Parallel Computing*, pp. 1033–1036, 2011.

[48] "Antutu benchmark." http://www.antutu.com/en/Ranking.shtml.

[49] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 143–158, IEEE, 2010.

[50] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 315–328, ACM, 2008.

[51] A. M. Azab, P. Ning, and X. Zhang, "Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 375–388, ACM, 2011.

[52] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Pixelvault: Using gpus for securing cryptographic operations," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1131–1142, ACM, 2014.

[53] M. Lindemann, R. Perez, R. Sailer, L. Van Doorn, and S. W. Smith, "Building the ibm 4758 secure coprocessor," *Computer*, vol. 34, no. 10, pp. 57–66, 2001.

[54] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pp. 367–378, IEEE, 2015.

[55] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang, "Protecting private keys against memory disclosure attacks using hardware transactional memory," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 3–19, May 2015.

[56] *CoreLink Level 2 Cache Controller L2C-310, Technical Reference Manual*, June 2012.

[57] Advanced Micro Devices, Inc., *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 15h Processors*, Jan 2013.

[58] "Everything you wanted to know about soc memory." http://www.low-powerdesign.com/Everything_You_Wanted_to_Know_About_SOC_Memory.pdf.

[59] *i.MX53 Multimedia Applications Processor Reference Manual*, June 2012.

[60] *i.MX 6Dual/6Quad Applications Processor Reference Manual*, July 2015.

[61] *ARM Cortex-A9 Processor Technical Reference Manual*, June 2012.

[62] *Intel Data Direct I/O Technology (Intel DDIO) A Primer*, Feb 2012.

[63] *ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0*, July 2015.

[64] M. Henson and S. Taylor, "Memory encryption: a survey of existing techniques," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 53, 2014.

[65] G. Duc and R. Keryell, "Cryptopage: an efficient secure architecture with memory encryption, integrity and information leakage protection," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pp. 483–492, IEEE, 2006.

[66] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," *SIGARCH Comput. Archit. News*, vol. 41, pp. 253–264, Mar. 2013.

[67] 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 *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 2–13, ACM, 2008.

[68] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 265–278, ACM, 2013.

[69] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 81–96, 2014.

[70] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig, "Lockdown: Towards a safe and practical architecture for security applications on commodity platforms," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, (Berlin, Heidelberg), pp. 34–54, Springer-Verlag, 2012.

[71] P. Gilbert, L. P. Cox, J. Jung, and D. Wetherall, "Toward trustworthy mobile sensing," in *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, pp. 31–36, ACM, 2010.

[72] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 380–395, IEEE, 2010.

[73] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pp. 267–283, USENIX Association, 2014.

[74] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan, "Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 13–24, ACM, 2013.

[75] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. Abu Ghazaleh, and R. Riley, "Iso-x: A flexible architecture for hardware-managed isolated execution," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 190–202, IEEE, 2014.

[76] S. W. Smith and S. Weingart, "Building a high-performance, programmable secure coprocessor," *Computer Networks*, vol. 31, no. 8, pp. 831–860, 1999.

[77] B. Garmany and T. Müller, "Prime: private rsa infrastructure for memory-less encryption," in *Proceedings of the 29th Annual Computer Security Applications Conference*, pp. 149–158, ACM, 2013.