Hindawi
Security and Communication Networks
Volume 2019, Article ID 8690853, 12 pages
https://doi.org/10.1155/2019/8690853

Research Article

SoftME: A Software-Based Memory Protection Approach for TEE System to Resist Physical Attacks

Meiyu Zhang[1], **Qianying Zhang**[1,2], **Shijun Zhao**[3], **Zhiping Shi**[1,4] and **Yong Guan**[1,5]

[1]*College of Information Engineering, Capital Normal University, Beijing 100048, China*
[2]*Beijing Advanced Innovation Center for Imaging Theory and Technology, Beijing 100048, China*
[3]*Institute of Software Chinese Academy of Sciences, Beijing 100190, China*
[4]*Beijing Key Laboratory of Light Industrial Robot and Safety Verification, Beijing 100048, China*
[5]*Beijing Key Laboratory of Electronic System Reliability Technology, Beijing 100048, China*

Correspondence should be addressed to Qianying Zhang; qyzhang@cnu.edu.cn

Received 29 November 2018; Accepted 6 February 2019; Published 4 March 2019

Guest Editor: Sascha Uhrig

The development of the Internet of Things has made embedded devices widely used. Embedded devices are often used to process sensitive data, making them the target of attackers. ARM TrustZone technology is used to protect embedded device data from compromised operating systems and applications. But as the value of the data stored in embedded devices increases, more and more effective physical attacks have emerged. However, TrustZone cannot resist physical attacks. We propose SoftME, an approach that utilizes the on-chip memory space to provide a trusted execution environment for sensitive applications. We protect the confidentiality and integrity of the data stored on the off-chip memory. In addition, we design task scheduling in the encryption process. We implement a prototype system of our approach on the development board supporting TrustZone and evaluate the overhead of our approach. The experimental results show that our approach improves the security of the system, and there is no significant increase in system overhead.

1. Introduction

The development of the Internet of Things (IoT) [1, 2] is hailed as the third wave of world information development after computers and the Internet. Embedded systems play a major role in the information processing of IoT. Embedded systems emerged in the 1970s and have been applied to various fields. They have high requirements for security and real-time [3]. Human beings are exposed to more and more smart devices in our daily lives, most of which are embedded systems. These devices store valuable personal information, making them the target of attackers and resulting in software attacks and memory data leakage accidents from time to time. To enhance the security of embedded systems, ARM proposed the TrustZone [4] security extension to its CPU architecture, which provides resource access control and memory isolation to protect sensitive data. ARM TrustZone technology plays an important role in information protection and has important applications on mobile embedded devices [5–7]. It divides hardware resources into a secure world and a normal world and builds an isolated trusted execution environment (TEE) for applications to protect trusted applications from compromised operating systems and applications [8].

As the value of the data stored in embedded devices increases, effective physical attack methods have emerged. From a cost perspective, inexpensive physical attacks allow attackers to launch attacks more easily, making them more serious practical threats to embedded systems. Some embedded devices are easily lost, such as mobile phones and personal computers, and the working environment of some embedded devices is unsupervised, which makes embedded devices more vulnerable to physical attacks. Such attacks mainly refer to cold boot attacks, bus monitoring attacks, and DMA (direct memory access) attacks. A cold boot attack obtains data by attacking physical memory. A bus attack can obtain data transmitted on the bus, and a DMA attack allows an attacker to use DMA interface of the device to obtain data. Attackers extract and analyze the data in the memory

to make illegal profits, making research on the security of embedded systems critical. Unfortunately, TrustZone does not enforce memory encryption, so it cannot resist above physical attacks [9]. Therefore even if sensitive information is stored in physical memory protected by TrustZone, an attacker can gain valuable information through inexpensive physical attacks. A solution to protect TEE against physical attacks is required.

In response to these issues, we propose SoftME, an approach to protect the confidentiality and integrity of sensitive applications. In our approach, we use TrustZone technology to allocate the on-chip memory space to the secure world, and execute TEE OS on the on-chip memory to protect against cold boot attacks. On-chip memory communicates with the core via the on-chip bus, so it can resist bus attacks. We use data encryption to protect the security of data transmission and storage off-chip. More specifically, the data is encrypted on the on-chip memory before being written back to the off-chip memory. That is to ensure that the data is always in the form of ciphertext when transmitted off-chip. We also design task scheduling for encryption and task execution, enabling parallel execution of tasks on a single processor. Compared to existing solutions, our approach has the following advantages. On the one hand, our design is based on software, without the need for additional hardware support. On the other hand, our approach does not need to modify the applications, and they can be executed directly. We implemented our prototype on the development board supporting TrustZone and evaluated its performance through experiments. The experimental results show that our approach improves the security of embedded systems and there is no significant increase in performance overhead.

In summary, we make the following contributions:

(i) We propose SoftME, an approach that uses the on-chip memory to defend against physical attacks. This approach allocates the on-chip memory space to the secure world of TrustZone, with no additional hardware support and no need to modify applications.

(ii) We protect the confidentiality and integrity of off-chip data by authenticated encryption algorithm and guarantee the fairness of the encryption process and task execution process through task scheduling.

(iii) We implement our prototype system on a physical development board supporting TrustZone, and the experimental results show that our approach improves system security and does not significantly increase system overhead.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 introduces background knowledge. Section 4 discusses our threat models and assumptions. Section 5 describes our design in detail and the security analysis of our approach. The implementation is described in Section 6, followed by the performance evaluation in Section 7. Finally, Section 8 concludes this paper.

2. Related Work

An attacker can obtain sensitive information in an embedded device through physical attacks. In order to resist physical attacks, the academic field has proposed various security solutions. This section briefly introduces these solutions, which can be divided into hardware assistance and software enhancements.

2.1. Hardware Assistance. As early as 2000, the Stanford University Computer Systems Laboratory implemented an execution only memory in the form of the hardware that supported internal compartments, and the compartments could not access each other [10]. In addition, in [11], the authors proposed a hardware-based memory protection scheme which protected the integrity of a single-core processor and the confidentiality of multiprocessor shared memory. In [12], they described the hardware mechanism of the SecBus project. The project used a separate hardware module to implement encryption protection, designed to protect the memory bus against bus attacks. To achieve the on-chip memory encryption, the Dartmouth College research team designed the Bear microkernel operating system [13, 14]. The Bear operating system provided an encryption mechanism using a security-enhanced processor to process data that appeared outside of the processor. In [15], the authors used a dedicated hardware detector to detect and prevent malicious attacks.

Hardware-assisted protection scheme has an advantage in performance; however, the disadvantages of integrating hardware components are that they occupy the space of embedded devices and increase energy consumption. Our approach relies on the on-chip memory which is available on many embedded development boards, and our approach is based on software to defend against physical attacks.

2.2. Software Enhancements. For embedded systems, sensitive data can be protected by encrypting memory. Hong D et al. proposed three different encryption methods for processing different sensitivity data. DynapoMP [16] was the first method to consider dividing the on-chip memory into different regions. For the problem of excessive memory encryption overhead, Papadopoulos P et al. implemented a secure memory allocator (s_malloc) [17] to allocate any size of memory from the heap dynamically, and any data written to this part of the memory would be encrypted. In addition, in [18, 19], the authors enhanced data security by redesigning the operating system. In [20], they proposed processor-memory bus encryption using the technique of locking cache. Similarly, in [9], Zhang N et al. also created a cache-based independent execution environment by locking the cache. In [21], the researchers proposed SecureME to hide data from compromised operating system and built a secure computing environment for applications. In [22], the authors proposed vTZ, a solution for virtualizing TrustZone. It provided a secure and isolated execution environment between guest TEEs. For physical attacks, Guan et al. proposed Copker [23], an encryption engine implemented inside the CPU, and they

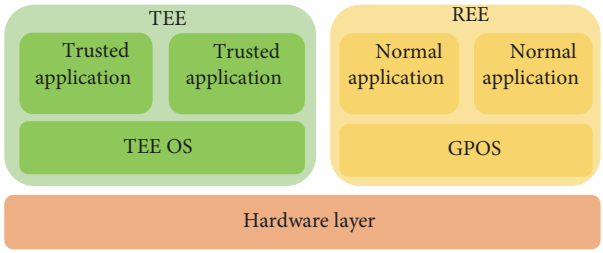

Figure 1: TEE architecture.

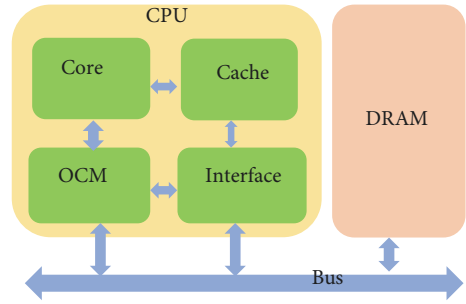

Figure 2: Embedded system based on the on-chip memory.

proved that it provides a secure service against cold boot attacks. In [24], the authors proposed Loop-Amnesia, a disk encryption technology that eliminated cold boot attacks. The mechanism is a kernel-based software design that provides protection for encryption keys in RAM and CPU.

Most of above designs require modifications to the operating systems and cannot be arbitrarily ported to other hardware platforms. One of the advantages of our approach is that we have implemented it in software without any modification to operating systems and it can be used to protect data within an acceptable overhead to the system.

3. Background

In this section, we will introduce the background knowledge related to our work. We will introduce the TEE architecture and ARM TrustZone first, and then we will discuss the on-chip memory architecture of ARM processor.

3.1. TEE and ARM TrustZone. Trusted execution environment (TEE) is proposed by GlobalPlatform. It is a trusted computing environment to provide security services to applications [25, 26]. TEE architecture is shown in Figure 1. The rich execution environment (REE) provides the execution environment for most functional applications, forming a dual operating system architecture with TEE [27]. In the dual operating system architecture, the GPOS and the TEE OS are executed on the same hardware platform, and hardware isolation based technology forms a secure execution environment for TEE OS. GPOS provides general functions to meet the functional requirements of the system, while TEE OS provides protection to trusted applications.

TrustZone is a set of secure hardware extension mechanisms for ARM processors to build an isolated computing execution environment for trusted applications [28]. ARM TrustZone technology reconfigures the processor, using the TrustZone Address Space Controller (TZASC) and TrustZone Protection Controller (TZPC) to divide the hardware resources into two separate parts, called the secure world and the normal world. In order to switch between the two worlds, ARM processor adds a new mode, monitor mode. The software running in this mode is called monitor, which is responsible for saving the context and restoring the state being switched while switching the worlds. In the CP15 coprocessor with the ARM processor, there is a Security Configuration Register (SCR) with an NS bit that indicates the current state of the processor. The secure world and the normal world can be switched by setting the NS bit in the monitor mode. If the NS bit is 1, it means that the current processor state is the normal world, and if it is 0, it means that the current processor state is the secure world. The processor can enter the monitor mode from the normal world by calling SMC (Secure Monitor Call) instruction or hardware interrupts.

3.2. On-Chip Memory Architecture. The application of the cache greatly improves processor performance, but it has unpredictable data access time. The emergence and widespread use of the on-chip memory solves this problem [29]. On-chip memory (OCM) is a general term for static random-access memory (SRAM) that is integrated into the chip for non-cache use. Compared with cache, OCM has the advantages of low power consumption, high performance, and small footprint. Banakar R et al. show that OCM reduces power consumption by 40% and the on-chip area by about 34% compared with the cache of the same capacity [30]. A typical embedded system with OCM is shown in Figure 2. The OCM and the cache communicate with off-chip memory through an off-chip data interface. Unlike off-chip memory, the on-chip memory and core communicate through the on-chip bus, so the on-chip memory is more secure than the off-chip memory.

In addition, OCM has a separate address space, and its address space is often mapped to the memory space. In this case, the entire address space is divided into two parts, OCM only occupies a small part, and the rest is still allocated to the off-chip memory. Therefore, the access from processor to OCM can be directly performed the same as the off-chip memory. Accordingly, the access mode of the on-chip memory greatly improves the speed of data reading and writing without going through the caching process. As a result, more and more embedded processors are beginning to integrate OCM on-chip to improve system performance.

4. Threat Model and Assumptions

To crack embedded systems to obtain valuable information, attackers have designed a variety of attack methods in the embedded system hardware level. In this section, we will

describe the threat model related to our design firstly and then introduce the assumptions of our design.

4.1. Threat Model. In our work, we focus on physical memory leaking attacks, such as cold boot attacks, bus monitoring attacks, and DMA attacks. For physical side-channel attacks, they require specialized protection techniques, and we do not consider this kind of attacks in this paper. We also do not consider physical attacks against the SoC, such as invasive attacks. This kind of attacks is targeted at information inside chips and requires specialized and expensive attack equipment, such as laser cutting system, microprobing station, oscilloscope, and focusing ion beam workstation. Only knowledgeable attackers can perform such attacks. So performing such attacks is quite expensive. The value of this equipment may far exceed the value of the targets being attacked. The attack process is also complicated, and even an experienced attacker may take several months, resulting in higher attack costs [31]. Therefore, attacks such as physical side-channel attacks, code injection attacks, and other complex physical attacks are out of the scope of our research.

4.1.1. Cold Boot Attacks. Cold boot attacks are a new type of physical attacks, which have become a part of many popular security threat models. In a cold boot attack, the attacker utilizes the data remanence effect of the memory to obtain the key and valuable information stored in the off-chip memory [32]. Experiments have shown that after the device is powered off, the data on the DRAM does not disappear immediately but will remain for a while [33]. Measurements of DDR1 and DDR2 demonstrate a correlation between temperature and RAM remanence, indicating that even if the surface temperature of the RAM module is slightly cooled by 10°C, the remanence effect is significantly extended [34]. One way to launch the cold boot attack is that an attacker utilizes this period of time to physically remove the DRAM from the target board and put it into the device that the attacker has prepared in advance to read the content in the DRAM. Another method is to restart the target board into an operating system controlled by the attacker to output memory content. Some research groups have successfully performed cold boot attacks on Android smartphones and retrieved private information such as encryption keys, address books, and photos from RAM [35].

4.1.2. Bus Monitoring Attacks. Buses are the crucial information transmission channels between various functional components, which have become the primary targets of attackers. Typical examples of such attacks are bus snooping attacks and bus tampering attacks. For example, an attacker can mount an FPGA board to the bus of an embedded system. By configuring and controlling the FPGA, the attacker can steal or even modify the data transmitted on the bus, thereby disrupting system execution.

Another typical example for bus monitoring attacks is in [36]. In the original Xbox game system, keys were stored in plaintext and transmitted over the South Bridge bus. The attackers exploited bus snooping and injecting to capture or insert information in the bus between system components to obtain the keys and decrypt the secure bootloader, thereby destroying the system trust chain. Subsequently, the attackers developed a low-cost chip that could be soldered to the game system bus, allowing users to bypass the security monitoring mechanism to play pirated games.

4.1.3. DMA Attacks. By configuring the devices that can use the DMA port, an attacker can bypass software security mechanisms and directly read the physical memory. One solution to such attacks is to utilize the IOMMU [37]. The operating system can program the IOMMU to limit the range of memory that the DMA device can access and even deny the DMA device access to memory. For devices equipped with IOMMU, IOMMU makes it impossible for malicious devices to access memory by DMA attacks. However, IOMMU is not available for every device. Fortunately, most ARM platforms support TrustZone, and DMA requests from compromised OS can be rejected to protect secure memory.

4.2. Assumptions. We assume that the ARM platform supports TrustZone technology and is equipped with the on-chip memory. On-chip memory is trusted to protect against physical attacks such as cold boot attacks, while off-chip memory and peripherals are untrusted. We also assume the existence of a device key in the SoC and it is trusted [38], such as KNOX's Device-Unique Hardware Key (DUHK).

5. System Architecture and Design

In this section, we will introduce the design of SoftME (a Software-based Memory Encryption protection approach) in detail based on the previously discussed techniques and assumptions. Finally, we will make a security analysis of our approach.

5.1. System Architecture. In our approach, we use the system architecture described in Figure 3. TrustZone divides hardware resources into two worlds, a secure world and a normal world. The memory of the secure world consists of two parts, one is the on-chip memory space and the other is a small portion of the DRAM. TEE OS and the monitor run on the on-chip memory, and the secure DRAM is used to store encrypted trusted tasks. Most of the space in DRAM is allocated to the normal world for GPOS. The monitor runs in monitor mode and is responsible for handling hardware interrupts and switching between the two worlds. Due to the resource isolation mechanism of TrustZone, GPOS cannot access devices and resources of the secure world, such as IO peripherals and memory, so TEE OS will not be affected by compromised operating systems and applications. For communication between the two worlds, we allocate a small piece of memory on the off-chip memory as shared memory. We design a task scheduler and a memory protection engine (MPE) in the TEE OS. The task scheduler is responsible for scheduling multiple tasks to ensure fair execution. The memory protection engine is used to decrypt or encrypt tasks while reading data from the on-chip memory or writing data

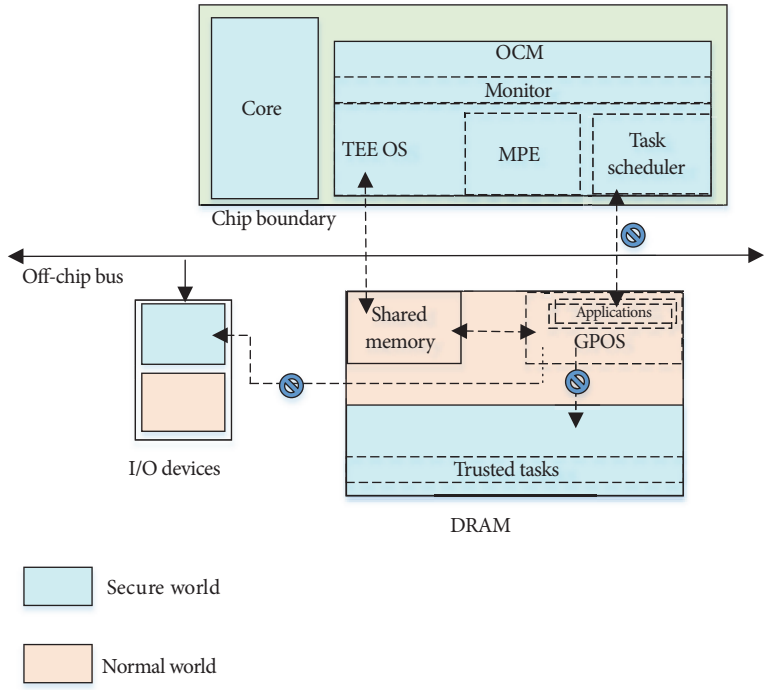

Figure 3: The architecture of SoftME.

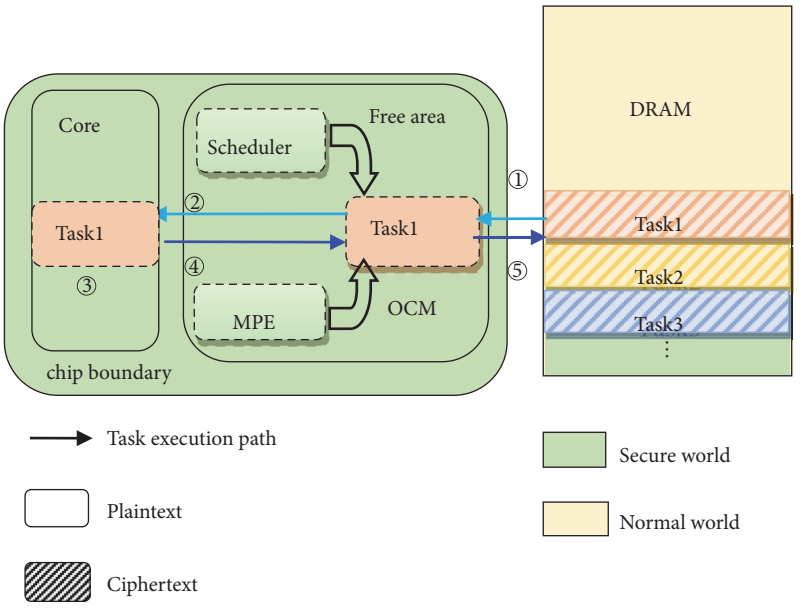

Figure 4: On-chip memory encryption workflow.

to the off-chip memory to ensure data confidentiality and integrity.

5.2. Memory Protection Engine Workflow. In order to prevent sensitive information from being stolen and tampered with, we must ensure the confidentiality and integrity of the data stored on the DRAM. Therefore, we design a memory protection engine in the TEE OS: when a task needs to switch from OCM to DRAM, the memory protection engine encrypts and protects the integrity of the data; when a task needs to be loaded from DRAM to OCM, the memory engine decrypts it and performs an integrity check. The specific process is shown in Figure 4. The execution process of a task in SoftME includes three phases: the loading phase, the

Table 1: Decryption process in the loading phase.

Get the unique key K_t.
Data transfer from DRAM to OCM.
1. Read the T and IV stored on the OCM.
2. $P = AE'(K_t, C, IV, T)$.

Table 2: Encryption process in the switching phase.

Initialization: derive a key from the device key:
K_t = HKDF (K_d, taskID).
Read plaintext from the OCM.
1. $(C, T) = AE(K_t, P, IV)$.
2. $IV = IV + 1$.
3. Store C to DRAM.
4. Store T, K_t and IV to OCM.

activation and execution phase, and the switching phase. Next we will give a detailed design description of these three phases.

(a) The Loading Phase. The loading phase corresponds to step 1 of Figure 4. This phase is to load the task from off-chip memory to the on-chip memory and decrypt it. Before the loading phase, we assume that the task on the off-chip memory has been encrypted. We use the authenticated encryption (AE) algorithm to protect the confidentiality and integrity of the task. The AE algorithm combines the message authentication code and the encryption algorithm to ensure the confidentiality and integrity of the data. Encrypting plaintext can ensure the confidentiality of the data and integrity authentication can verify whether data has been tampered with [39]. The generation of the unique key K_t for each task and the encryption process of the task will be described in the third phase. At this phase, we assume that the key K_t, ciphertext C, initialization vector (IV) IV, and the tag value T are already known. The tag value is the output parameter of the encryption process and is used to verify the integrity of data.

The description of the decryption process is shown in Table 1. The decryption algorithm is represented by the symbol AE', and the encryption algorithm is represented by AE. The ciphertext is the encrypted task read from the off-chip memory. The input parameters of the process are ciphertext C, IV, K_t, and T. If the authentication to the ciphertext is successful, then the plaintext is output; otherwise the symbol *FAIL* is returned. If the authentication fails, it indicates that the ciphertext has been tampered with by an attacker, and the task cannot be recovered. After obtaining the plaintext of the task, we put the task to the ready queue.

(b) The Activation and Execution Phase. The activation and execution phase corresponds to steps 2-4 of Figure 4. In this phase, a task is activated by the operating system and execution begins, just like the normal task execution process. In this phase, the task runs on the on-chip memory. On-chip memory is non-cached SRAM, so it can be read directly by the processor.

(c) The Switching Phase. The switching phase corresponds to step 5 of Figure 4. This phase includes encrypting the data after a task has been completed and storing the ciphertext to DRAM.

Each task has a unique key K_t. As mentioned earlier, we assume that the device key K_d exists and is trusted. We use K_d to derive K_t via the HMAC-based Key Derivation Function (HKDF) [40]. The HKDF algorithm is a key generation function based on the Hash-based Message Authentication Code (HMAC). There are two input parameters for HKDF, a device key and a message. We use the taskID created in the second phase as a message. Since each task has a unique taskID, the key K_t for each task is unique. The device key is highly confidential, and only the kernel has permission to read and operate it. Even if one task obtains the taskID of some other task, it cannot operate the device key, so it cannot obtain other task's key.

The encryption process in this phase is shown in Table 2. The input parameters of the process are the key value K_t, the plaintext P, and the initialization vector IV. The initialization vector is a random number with a fixed length, which is generally 16 bytes. The initialization vector value is incremented by 1 before each encryption. The GCM specification states that IV does not require randomness, but requires that the same key does not use the same IV [41]. We increment the IV by 1 to get different IV values. The plaintext is the task that the memory protection engine reads from the on-chip memory after the task execution is completed. The output values are the ciphertext C and the authentication tag T. After all these steps are completed, the ciphertext will be swapped to DRAM for storage. The tag value, key, and IV of a task will be stored on the on-chip memory.

5.3. Scheduler Workflow. In order to achieve fair scheduling for multitasking, we use the method of time slice polling to design the task scheduler. When a task is created, it will be given a priority, such as high priority, medium priority, or low priority. Tasks in the ready state with the same priority form a ready queue. Take the high-priority ready queue as an example. The workflow of the scheduler is shown in Figure 5. Assume that there are four tasks in the high-priority queue. Currently, task1 is in the running state, and the other three tasks are in the ready state. Suppose our time slice polling time is set to 1ms. During the time period of 0-1ms, task1 is running. At the time of 1ms (assuming that the time of state switching is tiny enough to be ignored), the scheduler sets the state of task1 from the running state to the ready state and places it at the end of the ready queue. Therefore, task2 gets the CPU resources, starts running, and so on. The task scheduler ensures that each task has the same fair execution time. Therefore, even if the CPU is single-threaded, at the high level, it still implements multiple tasks.

5.4. Security Analysis. Sensitive data is valuable to attackers. It is possible to be attacked by an attacker while the task

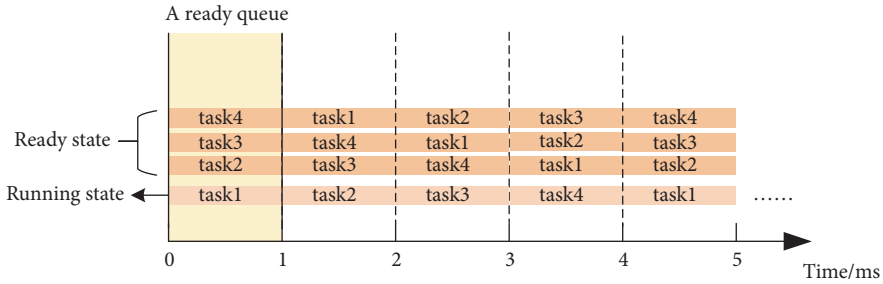

Figure 5: Workflow of the task scheduler.

is running and data is being transferred and stored. In this section, we will prove that SoftME is able to resist the physical attacks listed in the threat model.

Cold boot attacks. Operating systems and applications are vulnerable to cold boot attacks at runtime. In our approach, we run TEE OS and monitor on the on-chip memory. Compared to the off-chip memory, the on-chip memory has the following two features to prevent cold boot attacks. First, the cold boot attack needs to restart the device for subsequent attacks. For the on-chip memory, no matter for how long, after the device is powered off or restarted, the firmware on the board will initialize the on-chip memory and clear all its contents immediately [42], so the attacker could not get the confidential contents on the on-chip memory. In contrast, at room temperature, the off-chip memory can retain a portion of the content (0.1%) even after two seconds when the device is powered off. Second, the cold boot attack is launched when the bootloader of GPOS is started. At that time, the processor state has been the normal world, so the malicious code of the cold boot attack runs in the normal world and it cannot tamper with the on-chip memory, which has been partitioned to the secure world. To sum up, the on-chip memory will not be attacked by cold boot attacks.

Bus monitoring attacks. The on-chip memory is also secure against bus monitoring attacks, because sensitive data never leaves the on-chip memory and it is not transmitted over any exposed off-chip bus. However, the on-chip memory space is limited, it is impossible to store all sensitive data, and a part of sensitive data has to be stored in off-chip memory. For off-chip sensitive task, it can be protected by encryption. SoftME is designed to ensure that data does not appear in plaintext on the off-chip memory or on the bus. The authenticated encryption algorithm also generates a tag value while generating the ciphertext. This value is stored on the chip and used to perform integrity check on the data during decryption.

DMA attacks. On ARM platforms that support TrustZone, DMA attacks are also ineffective. Because DMA reads data directly from DRAM, the data we store on DRAM is processed by encryption. And the on-chip memory is allocated to the secure world, TrustZone will prevent illegal devices from accessing secure world memory through the DMA interface.

6. Implementation

In this section, we will detail the experimental environment and implementation of SoftME.

We implement our design on the Freescale i.MX6q SABRE Lite Board. It features Cortex A9 processor at 1 GHz per core, 1 GB of 64-bit wide DDR3, and 256K on-chip memory. The bootloader for hardware initialization and system boot is provided by onboard flash. The trusted operating system used for the prototype system is version 1.4.0 of TOPPERS/FMP, the general-purpose operating system kernel is Linux 3.10.53, and the monitor for world switching is SafeG 1.2.4. The serial port UART3 is selected as the information transmission port.

6.1. Memory Isolation. We build the SoftME architecture described in Figure 3 on the development board. Figure 6 shows the memory address arrangement of the prototype system. The memory system of the platform consists of two parts, on-chip memory and off-chip memory (DRAM). On-chip memory is all assigned to secure world, running the lightweight trusted embedded operating system TOPPERS/FMP. DRAM is divided into two parts, one for the secure world to store trusted applications and the other for normal world to run Linux. SafeG [43] is used as the monitor for world switching and runs on the on-chip memory. Since FMP, SafeG, and Linux storage space cannot be overlapped, we will store them separately in the memory area, where 0x00900000-0x0091FFFF is used by FMP, and 0x00920000-0x0092FFFF is used by SafeG. 0x12000000-0x4EFFFFFF of the off-chip memory space is used by Linux. The memory protection engine and task scheduler are running in FMP. The remaining size of the on-chip memory is calculated by the total size of the on-chip memory minus the size occupied by the TEE OS and the monitor. In our experimental platform, the total size of the on-chip memory is 256K, the size allocated for FMP kernel (including the memory protection engine and task scheduler) is 128K, and the size allocated for the monitor SafeG is 64K, so the remaining free space is 64K. Therefore, the maximum size of trusted tasks can be about 64K.

6.2. Port TEE OS to the On-Chip Memory. According to the above analysis, the off-chip memory is insecure and

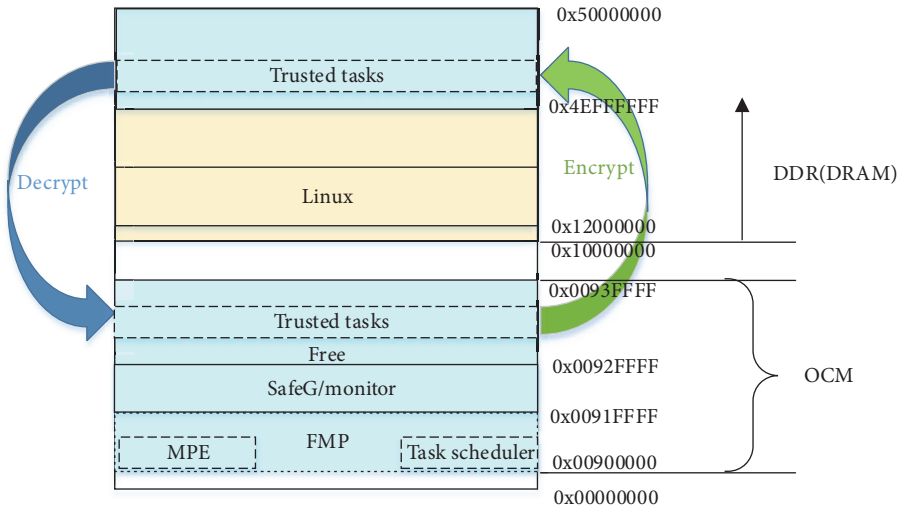

Figure 6: Memory address arrangement.

vulnerable to physical attacks. An important step in implementing SoftME is that the TEE OS should be executed on the on-chip memory. According to the i.MX 6Quad processors reference manual, the physical memory address range of the on-chip memory is 0x00900000-0x0093FFFF (Figure 6), and the entire on-chip memory region can be used freely after booting. We modified the FMP target-dependent configurable parameter value *TARGET_T_OS_START_ADDRESS* to be 0x00907000, which is in the on-chip memory address space range. In addition, the text base address and data base address of FMP are also modified to the physical address range of the on-chip memory. Therefore, after the system boots, FMP will be executed on the on-chip memory.

6.3. The Implementation of the Task Scheduler and the Memory Protection Engine. TOPPERS/FMP is a new generation trusted operating system kernel developed by Japanese TOPPERS project team that follows the μITRON 4.0 specification. μITRON is a real-time multitasking system specification that has become the Japanese industry standard. In terms of creating and activating tasks, we use APIs that comply with μITRON standards. The code sample for creating and activating the task scheduler is described in Table 3. We use the system static function, *cre_cyc ( )*, to create a task scheduler in the configuration file. An ID will be given to the task when the task is created. *sta_cyc ( )* is used for activating a task scheduler and the task state management function *irot_rdq ( )* is used for rotating task precedence. The task scheduler is activated in the TEE OS application to be available.

The role of the memory protection engine is to process the sensitive data on the DRAM. After executing on the on-chip memory, the task data is encrypted and then written back to the DRAM. The encryption key is derived from the device key. The code sample for creating and activating the memory protection engine is described in Table 4. In terms of implementation, we still use the functions of μITRON to create and activate tasks. *cre_tsk ( )* is used to statically create tasks in the configuration file, and *act_tsk ( )* activates tasks. The encryption algorithm we use is AES_GCM_128. We build a cryptography library by using Galois/Counter Mode (GCM). GCM is a kind of authenticated encryption algorithm with counter mode and message authentication code. Its performance evaluation has shown excellent performance on many platforms [44]. The value of parameter *mode* is either *encrypt* or *decrypt*. *encrypt* indicates that the process is an encryption process, and *decrypt* indicates that the process is a decryption process.

Table 3: Code sample of the task scheduler.

```
/* Create a task scheduler */
cre_cyc (taskid);
/* Implementation */
void task_scheduler (void)
{
irot_rdq (TASK_PRIORITY);
}
/* Activate the task scheduler */
sta_cyc (taskid);
```

Table 4: Code sample of the memory protection engine.

```
/* Create a memory protection engine */
cre_tsk (taskid);
/* Memory protection engine implementation */
void engine (void)
{
......
mbedtls_gcm_crypt_and_tag (mode, K_t, IV, P/C);
......
}
/* Activate the memory protection engine */
act_tsk (taskid);
```

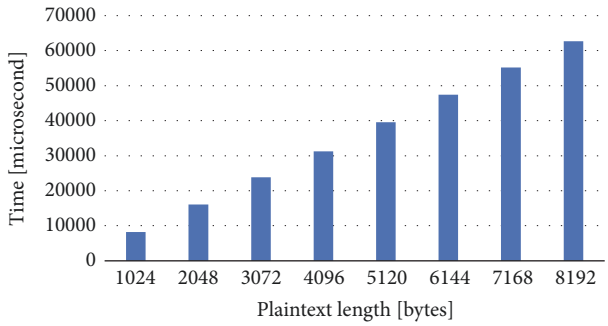

Figure 7: The overhead of the memory protection engine.

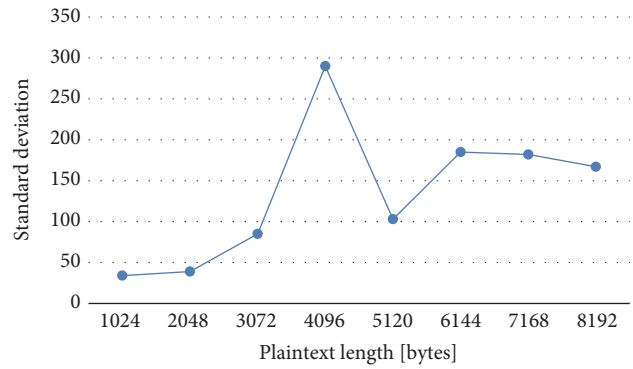

Figure 8: Standard deviation reflecting the degree of data dispersion.

Owing to the tiny size of the cryptography library we built, we can easily integrate the memory protection engine into FMP and port them to the on-chip memory. However, μITRON has a standard real-time kernel specification for any small embedded system design, including FMP, which results in some problems in the process of importing cryptographic algorithms in our experiment. More specifically, FMP developers use a custom function library specified by the standard; therefore if we directly imported the cryptographic algorithm to an FMP application, library conflict will be thrown. To address this problem, we replace the library file imported by memory protection engine with the FMP library file.

7. Experimental Results and Analysis

In this section, we evaluate the prototype system and give an analysis of the experimental results. In the experiment, we measured the overhead of the memory protection engine by encrypting different size of plaintext in the prototype system. Then we designed several trusted tasks for evaluation and analyzed the impact of the memory protection on the overhead of these tasks.

7.1. Code Modification. To coordinate the execution of the system, we need to patch the Linux kernel to support the execution of the monitor. The specific modifications are summarized as follows. We modified a total of 26 lines of code for 5 files in the bootloader, mainly some macro definitions related to switching the two worlds and the name of some startup file. A total of 2696 lines of code for 7 files are modified in Linux, mainly some conditional statements related to switching the worlds and SMC call to SafeG. Finally, we modified a total of 263 lines of code for two communication related configuration files in Linux to communicate with SafeG.

7.2. Overhead of Memory Protection Engine. First of all, we measured the overhead of the memory protection engine in the prototype system, with encrypting data whose length ranges from 1K to 8K. As demonstrated in Figure 7, the overhead of the memory protection engine increases almost linearly with the increase of the plaintext size.

Standard deviation is the most commonly used form of quantization that reflects the degree of dispersion of a set of data. Figure 8 shows the standard deviation of the data we collected. It can be seen that the overhead is variable and not very stable.

7.3. Overhead on Multitasks. In this section, we show the overhead of our approach in terms of the task execution time, the task switching time, and the task preemption time, and we analyze the experimental results separately.

To observe the impact of encryption on the simultaneous execution of multiple tasks, we designed six same tasks for evaluation. Since the time of each time slice of the FMP can be set to a minimum of 1 millisecond, in order to make the experimental results obvious, the execution time of each task we designed is longer than 1 millisecond.

For multiple tasks, we enable the tasks to be executed simultaneously by activating the task scheduler. To observe the impact of the memory protection engine on multitasks, we design two encryption strategies for the memory protection engine. The first strategy is that all the tasks are encrypted by only one memory protection engine. In this case, all tasks data cannot be encrypted at the same time and may generate waiting time. In the second strategy, multiple tasks are managed by multiple memory protection engines. That is, when the system creates a task, it also creates a memory protection engine accordingly. Owing to the tiny size of the memory protection engine, it will not have a significant impact on the system performance. In the experiment, each case is compared with the corresponding basic case tasks. These basic tasks are executed on the same experiment platform without the memory protection.

For the above three cases, we executed 1, 2, 4, and 6 tasks and measured their overhead separately. We have run 50 times for each case, and Figure 9 shows the experimental results and the comparison of the three cases. The overhead of a task execution includes task execution time, task switching time, and dispatch time of the task scheduler. It can be seen from the figure that the running time is almost linearly proportional to the number of tasks. Table 5 illustrates the overhead introduced by data encryption and shows the proportional relationship between strategy1/strategy2 and the basic case, respectively. For the first encryption strategy, the overhead of a task execution is increased by about 50%

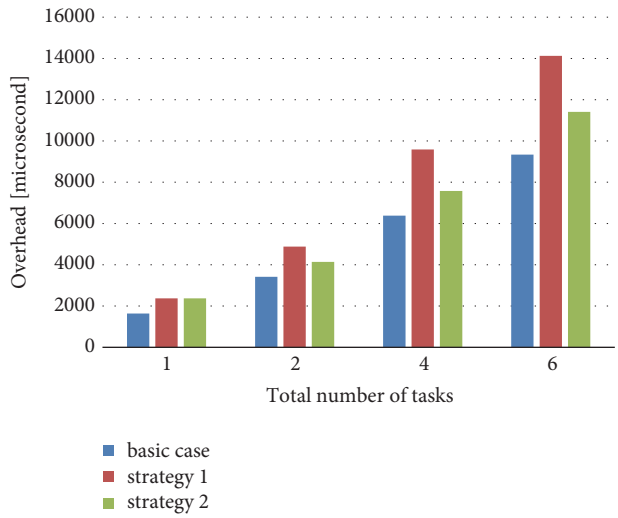

Figure 9: The overhead of different protection strategies.

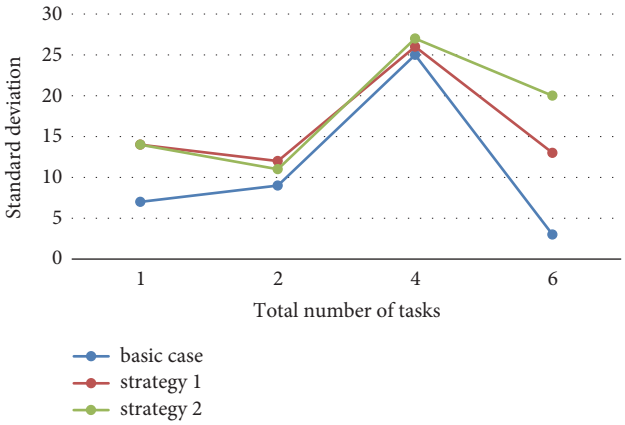

Figure 10: Standard deviation reflecting the degree of data dispersion.

compared to the basic case. When multiple tasks are executed, the advantages of the second strategy are obvious, since the overhead increases only about 20%.

Figure 10 lists the standard deviation of the overhead. Overall, these three cases are more stable than the overhead of the memory protection engine shown in Figure 8. We think it is because the execution time of the task hides the overhead of the memory protection engine. Among the three cases, the basic case is relatively stable, and we can see that although the second strategy has an advantage in terms of overhead, its stability is slightly worse than the first strategy.

As can be seen from above experimental results, in the embedded real-time operating system, ensuring the security of the tasks will inevitably affect the real-time task. Therefore, for a task requiring high real-time performance, we keep it on the on-chip memory. Therefore, the task does not need to be decrypted when it runs again. We measured the task preemption time and the task switching time for multitasking on the on-chip memory. The task preemption time is the time required for a high-priority task to preempt a low priority

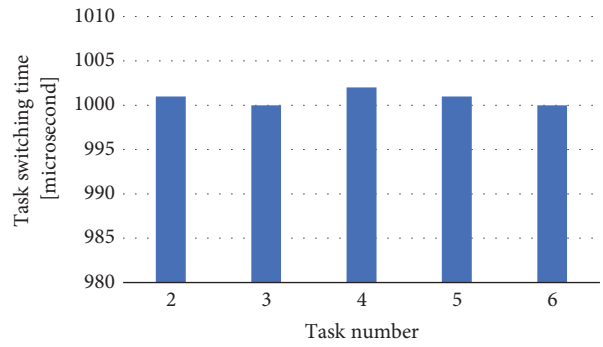

Figure 11: The task switching time.

Table 5: The proportional relationship between the two strategies and the basic case.

Number of tasks\ strategies	Strategy 1	Strategy 2
1	1.45	1.45
2	1.43	1.21
4	1.50	1.19
6	1.51	1.22

Table 6: The task preemption time.

	Average time/μs
Linux	8.56
basic case	0.86
strategy1	0.87
strategy2	0.86

task. It includes the time to save the context of the preempted task and the time to resume the context of the high-priority task. We measured the task preemption time in the Linux kernel, the FMP kernel without the memory protection, the FMP kernel running strategy1, and the FMP kernel running strategy2, respectively. The evaluation tasks are two tasks with different priorities, task1 and task2. The priority of task1 is set high, and the priority of task2 is set low. Task1 is activated in task2, so task1 can preempt task2. Using the system timer, the time before activating task1 and the time before running the first instruction of task1 are recorded, respectively. We run the experiments 100 times, and the average times are shown in Table 6. It shows that the FMP kernel has an absolute advantage over Linux in processing real-time tasks, and it can be seen that SoftME has no effect on the task preemption time.

The task switching time is the time spent on switching one task to another task. We activated six tasks at the same time and recorded the time when the first instruction of each task was invoked, denoted by t1 to t6. So the time spent on switching from task1 to task2 is t2-t1, the time spent on switching from task2 to task3 is t3-t2, and so on. The experimental result is shown in Figure 11. In Section 5.3, we configured the task switching time to be 1ms. It can be seen from the figure that the actual time is more than 1ms, which

is due to the need to save and restore the context when the tasks are switched.

8. Conclusions and Future Work

IoT devices have the disadvantage of being unsupervised, making them vulnerable to physical attacks. We propose SoftME, an approach for protecting trusted tasks against physical attacks. We use the on-chip memory to protect TEE OS and design a memory protection engine to protect the confidentiality and integrity of the tasks stored on the off-chip memory. We implemented SoftME on the physical development board. Finally, the experimental evaluation shows that the memory protection introduces an overhead of about 20%, which is within acceptable limits.

However, for a single-core embedded system, encryption will have a negative impact on the execution of real-time tasks. Multicore architecture has replaced single-core systems in many areas and has become the mainstream of embedded systems. In a multicore architecture, multiple tasks can be executed in parallel, with more parallel computing power, lower clock frequency, lower power consumption, and higher efficiency. Therefore, in future work, we can allocate a dedicated core for the memory protection engine, and other cores are responsible for executing tasks. This design will make use of the parallel computation of multicore to reduce the overhead caused by encryption and decryption, thereby improving the performance.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was supported by the National Key R & D Plan (2017YFB1301100), the National Natural Science Foundation of China (61602325, 61802375, 61702348), the Project of Beijing Municipal Education Commission (KM20190028005), the Project of the Beijing Municipal Science & Technology Commission (LJ201607), and Capital Normal University Major (key) Nurturing Project.

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): a vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] F. Wortmann and K. Flüchter, "Internet of things: technology and value added," *Business & Information Systems Engineering*, vol. 57, no. 3, pp. 221–224, 2015.

[3] P. Koopman, "Embedded system security," *Computer*, vol. 37, no. 7, pp. 95–97, 2004.

[4] P. Varanasi and G. Heiser, "Hardware-supported virtualization on ARM," in *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys '11)*, ACM, July 2011.

[5] N. Santos, H. Raj, S. Saroiu et al., "Using ARM trustzone to build a trusted language runtime for mobile applications," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 67–80, 2014.

[6] J. Y. Hwang, S. B. Suh, S. K. Heo et al., "Xen on ARM: system virtualization using xen hypervisor for ARM-based secure mobile phones," in *Proceedings of the 5th IEEE Consumer Communications and Networking Conference (CCNC'08)*, pp. 257–261, IEEE, Las Vegas, Nev, USA, January 2008.

[7] S. Yoo, Y. Liu, C. H. Hong, C. Yoo, and Y. Zhang, "Mobivmm: a virtual machine monitor for mobile phones," in *Proceedings of the 1st ACM Workshop on Virtualization in Mobile Computing*, pp. 1–5, ACM, June 2008.

[8] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: what it is, and what it is not," in *Proceedings of the 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015*, pp. 57–64, IEEE, Finland, August 2015.

[9] N. Zhang, K. Sun, W. Lou et al., "Case: cache-assisted secure execution on ARM processors," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy, SP 2016*, pp. 72–90, IEEE, USA, May 2016.

[10] D. Lie, C. Thekkath, M. Mitchell et al., "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[11] W. Shi, H. H. S. Lee, M. Ghosh et al., "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 123–134, IEEE Computer Society, 2004.

[12] L. Su, A. Martinez, P. Guillemin et al., "Hardware mechanism and performance evaluation of hierarchical page-based memory bus protection," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, p. 180, 2009.

[13] M. Henson and S. Taylor, "Beyond full disk encryption: protection on security-enhanced commodity processors," in *Proceedings of the International Conference on Applied Cryptography and Network Security*, pp. 307–321, Springer, Berlin, Germany, 2013.

[14] M. Henson and S. Taylor, "Attack mitigation through memory encryption of security enhanced commodity processors," in *Proceedings of the 8th International Conference on Information Warfare and Security (ICIW'13)*, D. Hart, Ed., pp. 265–268, USA, March 2013.

[15] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 12, pp. 1295–1308, 2006.

[16] D. Hong, L. A. D. Bathen, S.-S. Lim, and N. Dutt, "DynaPoMP: dynamic policy-driven memory protection for SPM-based embedded systems," in *Proceedings of the 6th Workshop on Embedded Systems Security*, ACM, Taiwan, 2011.

[17] P. Papadopoulos, G. Vasiliadis, G. Christou, E. Markatos, and S. Ioannidis, "No sugar but all the taste! memory encryption without architectural support," in *European Symposium on Research in Computer Security*, vol. 10493 of *Lecture Notes in*

Computer Science, pp. 362–380, Springer, Cham, Switzerland, 2017.

[18] L. Guan, P. Liu, X. Xing et al., "TrustShadow: secure execution of unmodified applications with ARM trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 488–501, ACM, USA, June 2017.

[19] W. Huang, V. Rudchenko, H. Shuang, Z. Huang, and D. Lie, "Pearl-TEE: supporting untrusted applications in trustzone," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pp. 8–13, ACM, Toronto, Canada, October 2018.

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

[21] S. Chhabra, B. Rogers, Y. Solihin et al., "SecureME: a hardware-software approach to full system security," in *Proceedings of the International Conference on Supercomputing*, pp. 108–119, ACM, 2011.

[22] Z. Hua, J. Gu, Y. Xia et al., "vTZ: virtualizing ARM trustzone," in *Proceedings of the 26th USENIX Security Symposium*, 2017.

[23] L. Guan, J. Lin, Z. Ma et al., "Copker: a cryptographic engine against cold-boot attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 742–754, 2018.

[24] 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.

[25] S. Pinto, T. Gomes, J. Pereira et al., "IIoTEED: an enhanced, trusted execution environment for industrial iot edge devices," *IEEE Internet Computing*, vol. 21, no. 1, pp. 40–47, 2017.

[26] Y. Fan, S. Liu, G. Tan et al., "One secure access scheme based on trusted execution environment," in *Proceedings of the 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (Trustcom/BigDataSE)*, pp. 16–21, IEEE, USA, 2018.

[27] J. Jang, S. Kong, M. Kim et al., "SeCReT: secure channel between rich execution environment and trusted execution environment," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, Calif, USA, 2015.

[28] D. Sangorrin, S. Honda, and H. Takada, "Dual operating system architecture for real-time embedded systems," in *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pp. 6–15, Brussels, Belgium, 2010.

[29] R. Tabish, R. Mancuso, S. Wasly et al., "A real-time scratchpad-centric OS for multi-core embedded systems," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2016*, pp. 1–11, IEEE, Austria, 2016.

[30] R. Banakar, S. Steinke, B. S. Lee et al., "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign, CODES 2002*, pp. 73–78, IEEE, USA, 2002.

[31] A. Tria and H. Choukri, "Invasive attacks," in *Encyclopedia of Cryptography and Security*, pp. 623–629, Springer, Boston, Mass, USA, 2011.

[32] S. F. Yitbarek, M. T. Aga, R. Das et al., "Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors," in *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 313–324, IEEE, 2017.

[33] M. Gruhn and T. Muller, "On the practicability of cold boot attacks," in *Proceedings of the 2013 8th International Conference on Availability, Reliability and Security (ARES)*, pp. 390–397, IEEE, USA, September 2013.

[34] J. A. Halderman, S. D. Schoen, N. Heninger et al., "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[35] M. Tilo, S. Michael, and F. C. Freiling, "Frost: forensic recovery of scrambled telephones," in *Proceedings of the International Conference on Applied Cryptography and Network Security*, pp. 373–388, Banff, AB, Canada, 2013.

[36] A. Huang, "Keeping secrets in hardware: the microsoft Xbox™ case study," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*, vol. 2523 of *Lecture Notes in Computer Science*, pp. 213–227, Springer, Berlin, Germany, 2002.

[37] P. Stewin and I. Bystrov, "Understanding DMA malware," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 7591 of *Lecture Notes in Computer Science*, pp. 21–41, Springer, Berlin, Germany, 2012.

[38] S. Zhao, Q. Zhang, G. Hu, Y. Qin, and D. Feng, "Providing root of trust for ARM TrustZone using on-chip SRAM," in *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, pp. 25–36, ACM, Scottsdale, Ariz, USA, November 2014.

[39] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*, T. Okamoto, Ed., vol. 1976 of *Lecture Notes in Computer Science*, pp. 531–545, Springer, Berlin, Germany, 2000.

[40] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," RFC Editor RFC5869, 2010.

[41] D. McGrew and J. Viega, "The Galois/counter mode of operation (GCM)," Submission to NIST Modes of Operation Process, 2004.

[42] P. Colp, J. Zhang, J. Gleeson et al., "Protecting data on smartphones and tablets from memory attacks," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 177–189, 2015.

[43] D. Sangorrin, S. Honda, and H. Takada, "Integrated scheduling for a reliable Dual-OS monitor," *Information and Media Technologies*, vol. 7, no. 2, pp. 627–638, 2012.

[44] D. A. McGrew and J. Viega, "The security and performance of the Galois/counter mode (GCM) of operation," in *Proceedings of the International Conference on Cryptology in India*, vol. 3348 of *Lecture Notes in Computer Science*, pp. 343–355, Springer, Berlin, Germany, 2004.