CryptMe: Data Leakage Prevention for Unmodified Programs on ARM Devices

Chen Cao[1](✉), Le Guan[1], Ning Zhang[2], Neng Gao[3], Jingqiang Lin[3], Bo Luo[4],
Peng Liu[1], Ji Xiang[3], and Wenjing Lou[2]

[1] The Pennsylvania State University, University Park, USA
{cuc96,lug14,pliu}@ist.psu.edu
[2] Virginia Polytechnic Institute and State University, Blacksburg, USA
{ningzh,wjlou}@vt.edu
[3] Institute of Information Engineering, CAS, Beijing, China
{gaoneng,linjingqiang,xiangji}@iie.ac.cn
[4] The University of Kansas, Kansas City, USA
bluo@ku.edu

Abstract. Sensitive data (e.g., passwords, health data and private videos) can be leaked due to many reasons, including (1) the misuse of legitimate operating system (OS) functions such as core dump, swap and hibernation, and (2) physical attacks to the DRAM chip such as cold-boot attacks and DMA attacks. While existing software-based memory encryption is effective in defeating physical attacks, none of them can prevent a legitimate OS function from accidentally leaking sensitive data in the memory. This paper introduces CryptMe that integrates memory encryption and ARM TrustZone-based memory access controls to protect sensitive data against both attacks. CryptMe essentially extends the Linux kernel with the ability to accommodate the execution of *unmodified* programs in an isolated execution domain (to defeat OS function misuse), and at the same time transparently encrypt sensitive data appeared in the DRAM chip (to defeat physical attacks). We have conducted extensive experiments on our prototype implementation. The evaluation results show the efficiency and added security of our design.

1 Introduction

Driven by the pressures of time-to-market and development cost, Internet-of-Things (IoT) manufacturers tend to build their systems atop existing open-source software stacks, notably the Linux kernel. Millions of IoT devices are running Linux kernel on ARM-based System-On-Chip (SoC), ranging from smart IP cameras, in-vehicle infotainment systems, to smart routers, etc. However, the swift prototyping process often comes at the cost of security and privacy. With full-blown software stacks, these devices often expose a much larger attack surface than we anticipated. Recent attacks against IoT devices have further indicated that our IoT devices are at higher and higher risk of being hacked.

M. Bailey et al. (Eds.): RAID 2018, LNCS 11050, pp. 380–400, 2018.
https://doi.org/10.1007/978-3-030-00470-5_18

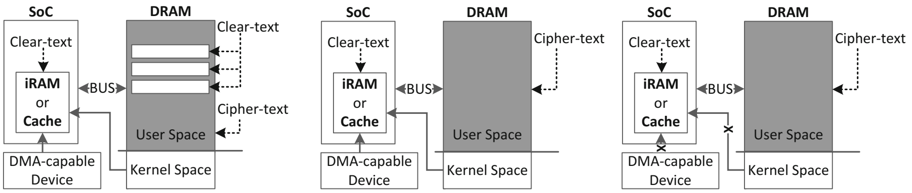
With a full-blown software stack deployed on IoT devices, sensitive data contained in programs often spread across all layers of the memory system [7]. A vulnerability in any layer can lead to the exposure of sensitive data. Unauthorized access to sensitive data residing on a DRAM chip is particularly serious because the data contained in the DRAM frequently include unprotected sensitive information (e.g., user credentials, video frames in an IP camera, Internet traffic with health data). Its exposure can be a major security concern for IoT device users.

In this paper, we aim to address two common types of DRAM-based memory disclosure attacks. First, in a software-based attack, private data in a program could be exposed to an attacker by misusing of benign OS functions or exploiting read-only memory disclosure vulnerabilities. For example, attackers can trigger normal OS functions such as coredump [22], hibernation/swap [12,21,34], and logging [7] to export otherwise isolated private memory to external storage. The second type of DRAM-based memory disclosure attack roots in the cyber-physical characteristic of IoT devices. Specifically, IoT devices are often deployed in diverse, and sometimes ambient environments; as a result, they are usually physically unmonitored. Attackers could physically access them and extract secrets contained in the DRAM chip [11]. Cold boot attack [16], bus-monitoring attack [10] and DMA attack [5] are quite common forms of physical attack. They can break the system even if the software is free of bugs.

Memory Encryption (ME) is a promising solution to address the aforementioned memory disclosure attacks. It operates on DRAM, and encrypts a portion or all of the address space of a program at runtime [19]. However, on one hand, ME solutions relying on hardware redesign increase the cost of the chip [24], and are not feasible for incremental Commercial Off-The-Shelf (COTS) defense deployment. On the other hand, existing general software-based ME solutions [8,13,29] all leave a small working set (memory that is currently being accessed) in clear-text to ensure the correct execution of a program. As a consequence, it is still possible for the working set to be exposed.

Gap Statement. An ME solution that really works on defeating the associated threats should protect both the non-working set memory and the working set memory *at all time*. In particular, it should have the following features: (1) The non-working set memory is encrypted; (2) The working set memory is in clear-text, but does not appear in the vulnerable DRAM. (3) The working set memory cannot be accessed by other software, including the OS. Unfortunately, to the best of our knowledge, a ME solution meeting all these requirements is still missing in the literature.

Software-based ME solutions can be classified into three types, as shown in Fig. 1. Cryptkeeper [29] and RamCrypt [13] belong to Type A (see Fig. 1a). In this category, most of the program data are encrypted while a small working set is left unprotected (e.g., four pages in RamCrypt) in the DRAM. As a result, Type A ME solutions are subject to both software and physical memory disclosure attacks. Type B solutions (see Fig. 1b) eliminate all the occurrences of clear-



(a) ME solution with partial data remaining in the DRAM

(b) ME solution without data remaining in the DRAM

(c) ME solution without data remaining in the DRAM and immune to software memory disclosure attacks

Fig. 1. Classification of ME solutions.

text program data in the DRAM chip by further protecting the working set by constraining them in the System-on-Chip (SoC) components such as `iRAM`) [18] or processor cache [8]. The SoC components are commonly believed to be much more difficult to attack compared with the DRAM chip [8]. Type B ME solutions are effective in defeating cold-boot attacks to DRAM chips. Unfortunately, the clear-text working set residing in the SoC components can still be exposed by software memory disclosure or DMA-capable devices.

As shown in Fig. 1c, Type C ME solutions disable both the OS kernel and DMA-capable devices to access `iRAM`. To implement a Type C ME system, a straightforward solution would be to further isolate clear-text program data in `iRAM`/cache from the OS kernel. In the ARM platform, the TrustZone architectural extension seems to be an ideal solution. With TrustZone, an ARM processor could run in two different execution domains – secure world or normal world. The OS in the normal world cannot access `iRAM` monopolized by the secure world. Therefore, if we execute the program in the secure world, and integrate existing type B ME solution, the problem seems to be solved. However, this is actually very challenging based on the following observations.

- **O1**: A legacy program runs in the same world with the OS. If the `iRAM` is a secure resource only accessible by the secure world, the legacy program in the normal world would simply crash; on the other hand, if the `iRAM` is designated to be a non-secure resource, the OS can still reveal the contents of the `iRAM`.
- **O2**: If we instead execute the legacy program in the secure world, there is no execution environment in the secure world. In particular, system services including system calls, interruptions, and page fault, etc., are all missing in the secure world.
- **O3**: To tackle the problem mentioned in **O2**, we could duplicate a full fledged OS in the secure world. However, the code base in the secure world will be inflated, making it prone to exploits.

Our Solution. In this work, we present CryptMe, the first type C ME solution for COTS ARM platforms. CryptMe addresses the aforementioned challenges by offloading a program in the secure world. Instead of employing a fledged OS to respond to the system service requests, we build a thin privileged layer in the secure world. The privileged layer does not provide system services itself, but forwards the requests to the OS in the normal world. By further incorporating type B ME solution, we ensure that both the non-working set and working set memory do not appear in clear-text in the DRAM chip, and the working set memory cannot be accessed by any software in the normal world.

Specifically, we protect sensitive data (called SenData) by encrypting all the anonymous memory segments (i.e., memory not backed a file, such as bss, heap, stack, and anonymously mapped memory segments) and private Copy-On-Write (COW) segments (such as data segment containing global and static variables). When the encrypted data are accessed, they are transparently decrypted in the `iRAM`. The program code in the DRAM chip is not protected. The key insight behind this is that the code segment of a program is usually publicly available so there is no need to protect its confidentiality. To further protect data in the `iRAM` from software attacks, CryptMe sets `iRAM` to be a secure resource. Therefore, even the OS kernel cannot access the data in it. To execute a protected process (called SenProcess), CryptMe offloads it to an isolated execution domain – TrustZone secure world, and a lightweight trusted execution runtime residing in the secure world is responsible for maintaining the execution environment of the process (such as setting up page tables). In summary, CryptMe ensures that clear-text program data *only* exists in `iRAM`, and we restrict accesses to `iRAM` from the Linux kernel by TrustZone configuration.

In summary, we made the following contributions.

- We have designed CryptMe, an ME system that prevents the clear-text sensitive data of *unmodified* programs from leaking to the DRAM for ARM-based IoT devices.
- CryptMe is the first ME system that is able to tackle both physical memory disclosure attacks and software attacks, including misuse of benign OS functions and real-only memory disclosure attacks.
- We have implemented CryptMe prototype on a Freescale i.MX6q experiment board. Security validation shows that CryptMe effectively eliminates all the occurrence of private program data in the DRAM, and thwarts software-based memory disclosure attacks.

2 Background

2.1 Memory Disclosure Attack

Though full system memory encryption has been a topic of interest, the privacy concerns for memory disclosure have not been a real threat until demonstrations of hardware-based memory disclosure attacks [5,10,16]. DMA capable devices such as Firewire were leveraged to read system memory [5]. Since DMA engine is independent of the processor, and directly talks to the DRAM chips, as long as the device is powered on, all the DRAM contents can be read out. In [16], Halderman et al. transplanted the memory chip of a laptop onto a different one where there was no software protection on the physical memory. Using a simple dust blower to keep the memory chip cool, it was possible to extract almost all of the information from the memory. The significance of this attack is that it can bypass all the software system protections. The remanence effect of DRAM was also exploited in [6,26] to launch cold-boot attacks to smartphones, where the system is rebooted into a rouge OS to bypass the memory protection. For advanced adversaries, it might even be possible to snoop the communication between the CPU and the DRAM [10].

Memory disclosure can also occur due to misuse of legitimate OS functions or passive read-only memory disclosure attacks. For example, the memory dump function is a very useful feature in modern OSes. A core dump image provides valuable information about the execution state when a crash happens which helps developer identify the crash point. However, attackers exploited this feature to dump sensitive data of a process [22]. Taking advantage of read-only memory disclosure vulnerabilities, the authors in [17] successfully exposed the private keys of an OpenSSH server and an Apache HTTP server.

2.2 TrustZone

TrustZone is a secure extension to the ARM architecture, including modifications to the processor, memory, and peripherals [35]. Most ARM processors support this security extension. TrustZone is designed to provide a system wide isolated execution environment for sensitive workloads. The isolated execution environment is often called *secure world*, and the commodity running environment is often referred to as the *normal world* or the *rich OS*. Different system resources can be accessed depending on the world of the process. In particular, the *Security Configuration Register (SCR)* in the *CP15* co-processor is one of the registers that can only be accessed while the processor is in the secure world. *NS (non-secure)* bit in the SCR controls the security context of the processor. When the bit is set, the processor is in the normal world. When the bit is clear, the processor is in the secure world.

One of the most important components in a TrustZone-based system is *TrustZone Address Space Controller (TZASC)*. Registers of TZASC are mapped into the physical address of the SoC, and can be accessed via memory operations. Access policies for different physical memory regions can be programmed via these registers. With these controls, secure world code can control whether a memory region can be accessed from both secure and normal worlds, or can only be accessed from secure world. For other peripherals, such as `iRAM`, different SoC manufactures implement different components to configure their access policy. In a typical implementation, a *Central Security Unit (CSU)* is used by trusted secure world code to set individual security access privileges on each of the peripheral.

3 Threat Model and Security Requirements

3.1 Threat Model

CryptMe is designed to prevent the sensitive data of a running program from being leaked into DRAM chip or other peripherals. The threats considered in this work, include (a) misused *benign* OS functions such as swap, hibernation, and core dump, (b) *passive read-only* memory disclosure attacks, and (c) *malicious* physical attacks targeting the DRAM chips.

We assume a benign OS kernel that runs in the normal world of a TrustZone-powered device. That is, basic OS services, such as task management, memory management and execution environment maintenance, etc. are trusted. We do not assume a compromised OS kernel. Otherwise, the process can be manipulated arbitrarily. We assume orthogonal solutions to ensure the integrity of the Linux kernel [4].

The OS is also assumed to correctly implement supplementary functions to improve efficiency (e.g., swap, hibernation), and to facilitate program analyses (e.g., core dump). However, once misused, these functions can be exploited to leak sensitive data, because they have the capability to access the whole address space of a process. There seems to be a countermeasure to deal with this issue – disabling these OS functions. However, many of them are indispensable in modern OSes. Once disabled, the whole system will be significantly affected. For example, disk swap is the key technique to support virtual memory. Without it, the system could quickly run out of memory.

The attacker could also exploit passive read-only memory disclosure attacks. When exploiting these read-only attacks, attackers often do not need to compromise the kernel to gain control flow and manipulate critical data structures. Therefore, active monitoring techniques (e.g., kernel integrity checking) cannot detect such "silent" data leakages. For example, in [17], the authors exploited two kernel vulnerabilities [27,28] to successfully extract private keys used in OpenSSH and Apache Server in several minutes. According to a statistics, this kind of "Gain Information" vulnerability contributes 16.5% of all Linux vulnerabilities as of Mar. 2018 [9].

We assume attackers are able to launch physical attacks to expose DRAM contents, bypassing the process isolation enforced by the OS. In a cold boot, the attacker is capable of dumping the entire DRAM image of a running device by rebooting it into another malicious OS from an external storage [16,26]. In DMA attacks [33], a malicious peripheral device is utilized to directly read out memory contents by issuing DMA requests. Moreover, an advanced attacker might even be able to eavesdrop data transmission between the DRAM chips and the processor by monitoring the memory bus [10].

The protected program itself must be trusted. That is, we assume a SenProcess never leaks SenData out of its private memory segments by itself, either intentionally or unintentionally. Since our protection is built on top of ARM TrustZone, we also assume the correctness of TrustZone implementations. The privileged codes of CryptMe running in the TrustZone secure world are

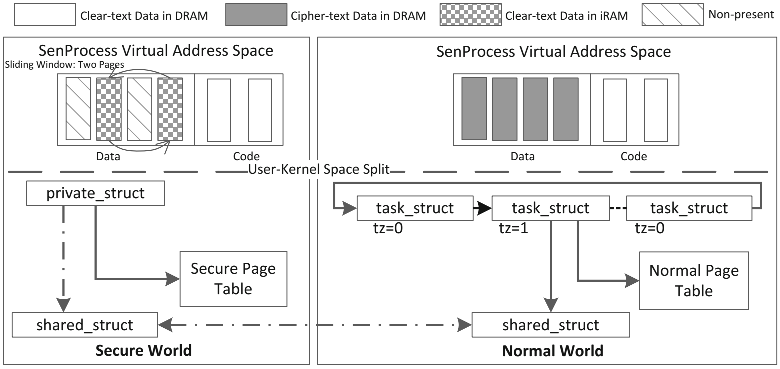


Fig. 2. CryptMe overview with a sliding window size of two pages

assumed to be free of vulnerability, as well as the trusted boot process enabling the TrustZone-based hardware memory control. In the design and implementation of CryptMe, we keep the privileged code base small (5.8K Lines Of Code (LOC), in the prototype system), so it is possible to formally verify its correctness. Lastly, side-channel attacks are out of the scope in this paper.

3.2 Security Requirements

Based on the threat model, we formalize the problem into the following security requirements that CryptMe aims to meet.

R1. The DRAM chip does not contain any clear-text SenData.
R2. The clear-text SenData is constrained in the on-chip `iRAM`, which can only be accessed by the secure-world code.

Software-based memory disclosure attacks are thwarted by the combination of **R1** and **R2**. In addition, meeting **R1** keeps SenData immune to cold-boot attacks and bus-monitoring attacks, while meeting **R2** prevents DMA attacks.

4 Design

This section describes the design of CryptMe. We start with an overview of the proposed system, then expand on several key techniques. We show how CryptMe supports offloading CryptMe-enabled SenProcesses to an isolated execution environment in the TrustZone secure world, and how page tables in this isolated environment are maintained. Finally we present the protections that CryptMe provides for the offloaded SenProcesses– encryption and isolation.

4.1 Overview

In CryptMe, a Linux OS runs in the normal world, while protected SenProcesses run in the secure world. As shown in Fig. 2, like any other processes in a Linux system, each SenProcess is referenced by a `task_struct` data structure in the normal OS. In fact, the `task_struct` of a SenProcess is no different from normal ones except for a newly added flag (`tz`) and a world-shared memory buffer (`shared_struct`). The flag identifies a process as a SenProcess while the shared buffer is used to exchange critical information (such as page table updates) between the two worlds.

Each SenProcess is still created, maintained, and scheduled by the normal OS, but executed in the secure world. The normal OS is customized so that just before a SenProcess is to return to user space, an `smc` instruction is issued to transfer the control to the secure world. In the secure world, there is a piece of *Secure Privileged Code* (SPC) that is responsible for maintaining the execution environment of a SenProcess by exchanging context information with the normal OS. Each SenProcess has its own `private_struct` that stores its hardware context, and `shared_struct` that is shared with the normal OS to enable data exchange.

When the SenProcess is executed in the secure world, its working data set is kept in clear-text in the `iRAM`, which is not accessible by the normal OS. For each SenProcess, SPC keeps a sliding window of `iRAM` pages for the working set. If the working set of a SenProcess exceeds the threshold assigned to it, SPC encrypts the oldest page in the window and copies it to the corresponding DRAM page, and then assigns the freed `iRAM` page to the virtual address that triggers the page fault.

A SenProcess has separate page tables in each world. Normal world page table is maintained by the normal OS with a customized page fault handler. It serves as a template for the *Secure Page Table* in the secure world. In both page table settings, the clear-text code segment is backed by the same DRAM pages, which CryptMe takes no effort to protect. However, SenData, which normal world page table maps to DRAM pages, is encrypted. SenData contained in the sliding window in `iRAM` is decrypted to keep the SenProcess runnable in the secure world, as shown in Fig. 2.

CryptMe employs the on-chip hardware-based cryptographic engine to accelerate AES computations. An AES key is generated randomly when a new SenProcess is about to be created. It is kept in a dedicated `iRAM` page shared by all the SenProcesses. The round keys and intermediate values generated during encryption/decryption are all constrained in this page, therefore, the key materials enjoy the same level of protection with that provided for SenData.

4.2 Executing in the Secure World

This section describes how a SenProcess gets offloaded to execute in the secure world. This is the prerequisite to enforce other security measures that will be discussed later. Since the secure world and the normal world are logically separated, SPC has to maintain the essential execution environments for SenProcesses to run in the secure world. In this section, we introduce a naïve code offloading mechanism, in which the normal-world page table and secure-world page table share the same set of page table entries. As a result, SenProcess code runs in the secure world, while all the memory references are routed to DRAM pages that both worlds can access. In Sect. 4.3, we show how to improve this naïve design to encrypt SenData that appear in the DRAM. Then, in Sect. 4.4, we further describe how to deprive the Linux OS kernel and other peripheral devices of the privilege to access clear-text SenData in `iRAM`.

Code Offloading. CryptMe supports memory encryption on a per-process basis. To start a SenProcess, the user land loader invokes a dedicated system call, similar `execve`, which marks the process in its `task_struct`.

With the capability to identify a SenProcess, the kernel is further instrumented to invoke an `smc` instruction whenever a SenProcess is about to be scheduled to run in user space. The `smc` instruction transfers control flow to the monitor mode in secure world, where the monitor mode code handles world switch, and invokes SPC to restore the hardware context of the SenProcess and execute it in the user space in secure world.

System Services. SenProcess in the secure world may incur exceptions during execution. When this happens, the SenProcess traps into SPC. To keep the code base of SPC small, SPC forwards all of them directly to the normal world OS kernel. In ARM platform, system calls are requested by the `swi` instruction, which traps the processor in the privileged `SVC` mode. Other exceptions such as interrupt and page fault trap the processor to the corresponding privileged CPU modes. To forward an exception to the normal world while keeping the normal OS oblivious of it, SPC needs to *reproduce* a hardware context as if the exception is triggered in the user space of the normal world. To achieve this, system registers indicating the context must be correctly set.

Re-producing Exceptions. Any SenProcess exception is first intercepted by the SPC. Because the monitor-mode code taking charge of world switches has ultimate privilege to access the resources of both worlds, it is possible to manually manipulate relevant registers that indicate the pre-exception context. Normally, these registers can only be set by hardware. With these registers manipulated, the system call handler in the Linux kernel can correctly parse the context information.

Page Table Synchronization. Each SenProcess in the secure world has its own page table. We instrument existing page fault handler in the normal Linux kernel to share the page table update information with SPC. This is based on the aforementioned exception forwarding mechanism. In particular, when a page fault exception is forwarded to the Linux kernel, it invokes its own page fault handler to populate the corresponding page in the normal world. Whenever the `set_pte_at` function is invoked, page table update information is duplicated in

Table 1. Cache and iRAM comparison

	Immunity to physical attacks	Capacity	Controllability	Continuous support	Intrusiveness
iRAM	✓	✗	✓	✓	✓
L2 cache	✓	✓	✗	✗	✗

the world-shared buffer shared_struct. The information includes the address of the page table entry, the updated value of the page table entry, the influenced virtual address, and other metadata. When the SenProcess is scheduled to execute in the secure world, SPC uses the shared information as a template to update the secure-world page table. In this way, SPC and the normal-world kernel maintain an identical copy of page table for each SenProcess.

4.3 Transparent Encryption

Barely offloading a SenProcess to the secure world does not gain any security benefit. This section describes how CryptMe enforces security requirement **R1**. That is, SenData appears in DRAM only as cipher-text.

To execute a process, the processor should always work on clear-text program data. In our design, a SenProcess runs with a clear-text working set that resides on on-chip memory unit, which is more expensive for an attacker to launch a physical attack. The rest of SenData is kept encrypted in the DRAM. Here, two commonly used on-chip memory units are processor caches and iRAM. We show the advantages and disadvantages of each option in the next paragraphs.

Selecting On-chip Memory. On-chip caches are small static RAM that are tightly coupled with a processor. It buffers recently accessed memories with very low access latency. In the recently shipped ARM SoCs, the capacity of a Level-2 (L2) cache can achieve several megabytes. When it loses power supply, all of its contents are lost. Therefore, in literatures, many solutions seek to defeat physical attacks to the DRAM chip using L2 caches [8,36].

iRAM is another on-chip memory that is more like a traditional DRAM chip. Most manufacturers integrate a 256 KB iRAM into their products to run boot code that initializes other SoC components. After that, all of its storage is free to use. During a reboot, the immutable booting firmware explicitly erases all the iRAM content [8]. Therefore, iRAM is also immune to cold-boot attacks. Table 1 summaries pros and cons for both L2 cache and iRAM.

Both options are suitable to defeat physical attacks. However, using cache has many drawbacks. First, even though cache can be used as SoC-bound memory storage, the dynamic nature of its allocation algorithm makes it difficult to lock its mapping to the physical memory address. Second, although many ARM processors support cache locking, this feature itself only benefits programs requiring customized cache allocation to maximize cache usage. As the size of cache is growing in each generation of processors, the need for customized cache use is diminishing. As a consequence, this feature is becoming obsolete in the latest generations of ARM processors such as Cortex-A57 [3]. Furthermore, cache is designed to ease the bottleneck at the slow memory operations. Monopolizing cache for security purpose can severely degrade the overall system performances. Therefore, in CryptMe, we choose `iRAM` to back the clear-text working memory.

Memory Encryption. Building atop the page table synchronization mechanism introduced in Sect. 4.2, SPC further differentiates the types of page table updates for a SenProcess. In particular, within the shared data structure `shared_struct`, a flag indicating the property of the corresponding fault page is added. The flag instructs SPC how to set up the page table – to duplicate the normal-world page table entry that points to an identical normal DRAM page (e.g., for a code page), or to allocate a new page in the `iRAM` (e.g., for an anonymous data page). In the latter case, SPC replaces the target normal-world DRAM page address with the newly allocated `iRAM` page address in the secure-world page table entry, and then maintains the mapping. Since the capacity of an `iRAM` chip is limited, SPC cannot meet all the page table requests of a SenProcess. We introduce a sliding window mechanism to address this problem.

Sliding Window. SPC assigns a dynamic number of `iRAM` pages to each SenProcess. Starting from the first available `iRAM` page, SPC keeps a circular index to the next available `iRAM` page. Page faults corresponding to SenData accesses continue to consume `iRAM` pages until the assigned pages are used up. In this case, the circular index points to the first `iRAM` page in the window. SPC then encrypts that `iRAM` page and copies it to the corresponding DRAM page. Finally, this `iRAM` page is assigned to be used for the newly occurred page fault request.

4.4 Disabling Access to the Sliding Window

We have ensured that no clear-text SenData would occur in the DRAM. However, privileged kernel can still read out any program data in the sliding window contained in `iRAM`. This flaw actually exists in all the existing software-based memory encryption solutions, such as Bear [18], RamCrypt [13], and Crypt-Keeper [29]. Moreover, it is possible that a local attacker issues DMA requests to `iRAM`. CryptMe addresses this threat by enforcing hardware-based access control to `iRAM`. More specifically, during booting, CryptMe configures the *CSU* available in TrustZone so that normal world code, including the Linux kernel, and any other peripherals, cannot access `iRAM`. This effectively enforces security requirement **R2**. That is, `iRAM` that holds clear-text SenData cannot be accessed by any entities other than the secure world code.

5 Implementation

We have implemented a full prototype of CryptMe on a Freescale i.MX6q experiment board which features an ARM Cortex-A9 processor with 1 GB DDR3

DRAM and 256 KB iRAM. Our implementation includes two parts. In the secure world, the implementation of SPC comprises around 5.3K LOC of C, and 0.5K LOC of assembly. In the normal world, we instrument the Linux kernel version 3.18.24 to be CryptMe-aware with 300 LOC of modification.

5.1 Secure World

The experiment board supports High Assurance Boot (HAB), a proprietary technology to ensure trusted boot. After power on, a proprietary boot ROM in the board executes to initialize critical system components and verify the integrity of the next stage image – in our case, the SPC. If SPC passes checking, it gets execution privilege in the secure world. Otherwise, the ROM will be reset.

To disable access to iRAM from DMA and the Linux kernel, SPC configures the CSU to set iRAM as a secure master. In our implementation, we achieve that by enabling the OCRAM_TZ_EN bit in register IOMUXC_GPR10, and setting access control policy in the low 8 bits of the CSU_CSL26 register in CSU[1]. Then SPC locks the configuration. As a result, any intentions to make modifications to the CSU configuration will trigger a system reboot, including SPC itself.

Finally, SPC hands the control to the boot loader in the normal world – uboot, which further boots the Linux OS.

5.2 Normal World

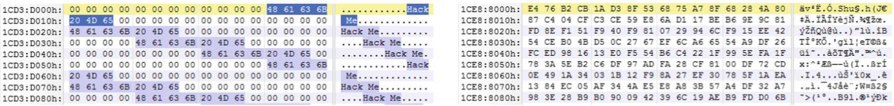
SenProcesses are still created and scheduled by the Linux kernel. We add a customized system call execve_enc to load a SenProcess. A process started with execve_enc has a tz flag set in its task_struct. We instrument the ret_to_user and ret_fast_syscall routines, so that whenever a SenProcess is about to return to user space, an smc instruction is issued to route the execution in the secure world. To run an unmodified program as a SenProcess, the user only needs to invoke a wrapper program with the path of the target program as a parameter. The wrapper program simply replaces itself with the target program by invoking the execve_enc system call.

5.3 Key Management and Encryption

When a SenProcess is created by execve_enc, the SPC invokes the on-board hardware-based random number generator to extract a 256-bit AES key anew. This key is used to protect all the SenData of this SenProcess. When the the process is terminated, the key can be safely discarded, because the anonymous SenData which it protects, do not persist across invocations.

The experiment board we use integrates Cryptographic Acceleration and Assurance Module (CAAM), which provides accelerated cryptographic computation, including AES, DES/3DES, RC4, etc. We employed CAAM to implement

[1] CSU_CSL is a set of registers only accessible in secure state that can set individual slave's access policy. Low 8 bits of CSU_CSL26 is marked as reserved in the manual of our experiment board, we found that it controls access to iRAM by experiments.



```
1CD3:D000h: 00 00 00 00 00 00 00 00 00 00 00 00 48 61 63 6B  ............Hack
1CD3:D010h: 20 4D 65 00 00 00 00 00 00 00 00 00 00 00 00 00   Me.............
1CD3:D020h: 48 61 63 6B 20 4D 65 00 00 00 00 00 00 00 00 00  Hack Me.........
1CD3:D030h: 00 00 00 00 48 61 63 6B 20 4D 65 00 00 00 00 00  ....Hack Me.....
1CD3:D040h: 00 00 00 00 00 00 00 00 48 61 63 6B 20 4D 65 00  ........Hack Me.
1CD3:D050h: 00 00 00 00 00 00 00 00 00 00 00 00 48 61 63 6B  ............Hack
1CD3:D060h: 20 4D 65 00 00 00 00 00 00 00 00 00 00 00 00 00   Me.............
1CD3:D070h: 48 61 63 6B 20 4D 65 00 00 00 00 00 00 00 00 00  Hack Me.........
1CD3:D080h: 00 00 00 00 48 61 63 6B 20 4D 65 00 00 00 00 00  ....Hack Me.....
```

(a) Image dumped with a native Linux Kernel.

```
1CE8:8000h: E4 76 B2 CB 1A D3 8F 53 68 75 A7 8F 68 28 4A 80  äv²Ë.Ó.Shu§.h(J€
1CE8:8010h: 87 C4 04 CF C3 CE 59 E8 6A D1 17 BE B6 9E 9C 81  ‡Ä.ÏÃÎYèjÑ.¾¶žœ.
1CE8:8020h: FD 8E F1 51 F9 40 F9 81 07 29 94 6C F9 15 EE 42  ýŽñQù@ù..)”lù.îB
1CE8:8030h: 54 CE B0 4B D5 0C 27 67 EF 6C A6 65 54 A9 DF 26  TÎ°KÕ.'gïl¦eT©ß&
1CE8:8040h: FC ED 98 16 13 E0 F5 54 B6 C4 22 1F 99 5E FA 1F  üí˜..àõT¶Ä".™^ú.
1CE8:8050h: 78 3A 5E B2 C6 DF 97 AD FA 28 CF 81 00 DF 72 CD  x:^²Æß—ú(Ï..ßrÍ
1CE8:8060h: 0E 49 1A 34 03 1B 12 F9 8A 27 EF 30 78 5F 1A EA  .I.4...ùŠ'ï0x_.ê
1CE8:8070h: 13 84 EC 05 AF 34 4A E5 E8 A8 3B 57 A4 DF 32 A7  .„ì.¯4Jåè¨;W¤ß2§
1CE8:8080h: 98 3E 28 B9 B0 90 09 42 39 6C 19 AE B9 FD D0 6B  ˜>(¹°..B9l.®¹ýÐk
```

(b) Image dumped when CryptMe is enabled.

Fig. 3. Physical memory image with and without CryptMe enabled.

a SoC bounded cryptographic service. Specifically, during an AES computation, all the sensitive data, including the original AES key, its key schedule, and intermediate results are redirected into a single reserved `iRAM` page. As a result, this page, together with plain-text SenData, has the highest protection level in our system. In CryptMe, we use AES-256 in CBC mode. The Initialization Vector (IV) is chosen as the virtual address of the encrypted page.

6 Evaluation

In this section, we evaluate CryptMe in both security and performance. In terms of security, we designed and conducted experiments to validate the security requirements **R1** and **R2** in Sect. 3.2. In terms of performance, we measured the overhead introduced by CryptMe compared with the base line in the native Linux environment. Our evaluation was performed on the same board and the same software environment as our prototype.

6.1 Security Evaluation

This section introduces several simulated attacks we designed to evaluate the security features of CryptMe.

Meeting Security Requirement ***R1***. Security requirement **R1** states that the DRAM chip contains no clear-text SenData. In order to obtain the contents of DRAM chip, we use the "memdump" utility to dump memory contents from the `/dev/mem` device file. To test the effectiveness of our system, we wrote a simple program which constantly writes a magic string ("Hack Me") into memory. Then we dump the whole DRAM image to search for this magic string.

Figure 3 depicts the results on the dumped images we obtained from the native Linux and CryptMe. The addresses displayed in these figures are the offsets from the beginning of the dump file. The beginning of this file represents the contents of the beginning of DRAM, which has an offset from the start of physical memory map, therefore, the real physical address is calculated by deducing this DRAM offset from the displayed file offset. Figure 3a shows the result from the native Linux kernel. Clearly, we were able to locate a bunch of magic strings in the dump image. Figure 3b shows the result we obtained when CryptMe is enabled. Throughout the searching, we did not find any occurrence of “Hack Me” string. This indicates that all the magic strings are encrypted in DRAM.

Meeting Security Requirement ***R2***. Security requirement **R2** states that on-chip iRAM cannot be accessed by any entities other than the secure-world software. To simulate an attack targeting iRAM, we wrote a kernel module that deliberately maps iRAM to the address space of a process using the vm_iomap_memory kernel function, and attempted to read the iRAM content in the normal world. The result shows that we can only obtain zero values, regardless of what we wrote into the iRAM. On the contrary, after we disabled hardware access control enforcement on iRAM as mentioned in Sect. 4.4, we were able to read out the data that the process wrote.

Defeating Attacks Misusing Legitimate OS Functions. In a software-based attack that misuses legitimate OS functions, the whole address space of a SenProcess is exposed. A kind of such attacks takes advantage of the coredump function which was originally designed to assist program analyses when a crash happens. In particular, the attacker deliberately crashes the target program, and it triggers a coredump operation which allows the OS to generate an image containing target process’s memory contents, CPU context etc., when the crash happens. As the image is stored in the persistent storage (i.e., flash chip in an IoT device), the attacker could easily read it out.

In order to simulate such an attack, we sent a “SIGSEGV” signal to the victim SenProcess to trigger a coredump after it writes a bunch of magic values (0xEF87AE12) into its anonymous memory segment. We got the coredump images of this process from the systems running with and without CryptMe enabled. As expected, we successfully found the target value in the image dumped from the native Linux system. On the contrary, we did not find any occurrence of 0xEF87AE12 in the image dumped when CryptMe is enabled throughout the searching process.

6.2 Performance Evaluation

To evaluate the performance overhead, we compare the benchmarks of programs in three system configurations. They are (1) **native** Linux system without modification, (2) CryptMe using the AES algorithm to **encrypt** pages being swapped, and (3) CryptMe using **plain** copy to swap pages. We first tested our system with the LMbench micro-benchmark [25] to measure the overhead introduced by world switches. This overhead is inevitable if we want to shield the iRAM from attacks. Next we tested our system with a self-written AES benchmark. This lightweight cryptographic primitive is frequently used in IoT devices. Finally, the performance of Nginx, a large complex web server is measured. Lots of IoT devices expose a web interface for users to access their functionality or to perform configuration changes to them. To better understand the introduced overhead, we designed experiments to measure the time consumption of different steps in the program execution.

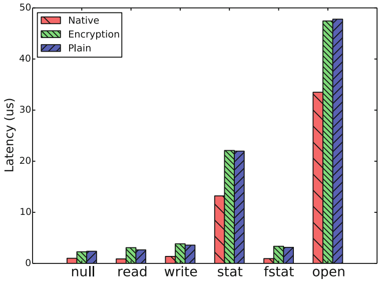


Fig. 4. System call latency.

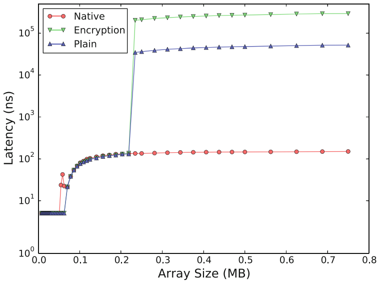


Fig. 5. Latency of memory reading with varying buffer sizes.

LMbench. SPC acts as an intermediate layer in-between the user space in secure world and kernel space in normal world. This design doubles the length of path to travel from user space to the Linux kernel and increases context-switch overhead. Therefore, we first report our results on the `lat_syscall` test, which measures the response time for various system calls.

Figure 4 depicts the results of `null`, `read`, `write`, `stat`, `fstat`, and `open` operations [25]. As shown in the figure, compared with the native Linux system, it takes CryptMe almost 3 times longer to complete `null` and `read` operations. However, such overhead is amortized in other non-trivial operations. For example, the performance overhead for the `open()/close()` system call is only about 1.5 times. Moreover, CryptMe with AES encryption and CryptMe with plain copy exhibit very similar performance. This is expected because a system call is not likely to trigger a page swapping between DRAM and `iRAM`.

`lat_mem_rd` is a program included in the LMbemch test-suite that measures memory read latency. It reads memory buffers with varying sizes from 512 B to 768 KB. Because the maximum working set is obviously larger than the sliding window of a SenProcess, `lat_mem_rd` effectively exposes and even enlarges performance overhead caused by CryptMe.

We explain the measured data as following. Since `lat_mem_rd` is a memory-intensive program, when the size of the buffer is small enough to be fit in the sliding window, very few pages need to be swapped in and out of the `iRAM`. As a result, no additional CPU cycles are needed. This is what we can see in Fig. 5 before the array size reaches 0.25 MB. At this stage, the three lines overlap with one another. When the buffer size exceeds that of the sliding window, old pages in the sliding window need to be swapped out to make room for new page requests. The introduced swapping operations indeed cause an abrupt performance degradation. Additional overhead can also be observed between CryptMe with encryption and CryptMe with plain copy. This is caused by the additional CPU cycles spent on the AES encryption.

Although the overhead introduced by CryptMe appears to be significant in this experiment, we would like to argue that: (1) such extremely memory-intensive use cases are very rare in real-world applications, especially in IoT devices. And (2) with the development of hardware technologies and reduced costs, commercial IoT devices on the market are often loaded with computing powers that are significantly beyond their needs.

Table 2. AES-128 throughputs with different numbers of threads (completed AES blocks per second)

Thread #	1	2	3	4	5	6
Native	62011	63832	63862	62847	62858	62863
Encryption	63187	64213	64256	63243	63268	64316

Table 3. Nginx performance (requests per second)

	Sliding window = 16	Sliding window = 32	Sliding window = 48
Plain	109.30	247.95	574.04
Encryption	23.60	72.26	571.32

AES Benchmark. We implemented an AES benchmark based on mbed TLS [2] library. It computes AES-128 for 500,000 times using different numbers of threads. As AES is a computation-intensive program with small memory footprint, Table 2 clearly shows that CryptMe incurs negligible overhead. Both CryptMe and native Linux complete around 63,000 AES block calculations per second regardless of the number of computing threads.

Nginx Web Server. We also measured the overhead of CryptMe when serving large complex programs. Many IoT devices provide their users with a web interface, through which the users are able to access the service or configure the device.

Table 4. Raw HTTP performance measurements (requests per second).

	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB	1024 KB
Native	655.70	625.64	633.62	604.30	513.94	434.26	310.75	208.74	124.36	71.14	39.20
Encryption	601.97	560.68	580.87	554.98	474.72	403.92	292.52	195.08	121.51	70.18	39.05
Overhead	1.09x	1.12x	1.09x	1.09x	1.08x	1.08x	1.06x	1.07x	1.02x	1.01x	1.00x

Nginx [31] is an open-source high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server. We used Nginx version 1.10.1 to run a HTTP web server, and used Apache benchmark [1] to measure the performance of the systems. The HTML file is the default 151 bytes welcome page, and the base line measured with native Linux system is 647.10 requests per second. In Table 3, we present the throughput of CryptMe under different sliding window sizes. In Table 4, we compare the HTTP throughput of CryptMe under 48-page sliding window size with native Linux system for different raw file sizes. When the sliding window is 48 pages, comparable performance is observed. Therefore, we would like to conclude that the overhead introduced by CryptMe is very acceptable, because of the redundant computing power in such systems. However, as the sliding window decreases, the overhead becomes non-negligible. It is clear that frequent page swapping causes the noticeable overhead. In the following, we present a break-down measurement of additional time consumed in world switching and page swapping.

Table 5. Break-down measurement of time consumed in each period.

Operation	Time (μs)
Context switch	2.27
Encryption/decryption & copy	326.32
PTE setup	7.01

Break-down Measurement. Based on the above experiment results, CryptMe is friendly to computation-intensive programs while exhibits ineligible overhead to memory-intensive programs. For memory-intensive programs, frequent page swapping is the key factor that influences the performance. In Table 5, we show a break-down measurement of the time spent on handling a page fault due to page swapping. Context switch is the time when completing a `getpid()` system call, which is drawn from Fig. 4. Note that this represents the minimum time for a world switch. Encryption/decryption & copy is the time spent on a encrypting/decrypting a page and copying it to normal/secure world. Note that a page swap invokes this operation twice; one for encrypting an old page into DRAM, and the other for decrypting a cipher-text page into `iRAM`. Finally, PTE setup measures the time for installing a page table entry in the secure world. It can be observed that cryptographic operation remains the dominating factor, which is the necessary price for the additional protection in memory encryption in general. However, many IoT devices are designed to be single purpose devices with limited functionality, therefore often do not require large working sets. This fixed cost for data encryption can be further reduced with more efficient hardware implementation of the cryptographic primitive.

7 Related Work

7.1 Memory Encryption

Many solutions on system memory encryption is motivated by the need to protect sensitive information stored in the memory [30]. With the rapid increase in speed and more sophisticated hardware-supported cryptographic function in modern processors, there has been recent efforts to realize practical software-based memory encryption on COTS hardware [8,13,14,18,29]. In particular, Cryptkeeper [29] and RamCrypt [13] implement ME on x86 platforms on a per-page basis with configurable security. Their implementation keeps a small set of decrypted working pages called sliding window. CryptMe also adopts the sliding window concept, but the decrypted working set is stored in the on-chip memory, which is protected from memory attacks [16]. In [14], hypervisor is used to encrypt kernel and user space code in guest operating systems, and the decrypted working set is configured to fit the cache. Bear [18] is a comprehensive ME solution that hides working set in the on-chip memory. However, this work focuses on a "from scratch" microkernel that does not fit commodity OS. Sentry encrypts sensitive Android application when the device is locked, and employs on-chip caches to support background applications [8]. This solution is not practical for applications at normal state because substantial performance slowdown is observed. All the aforementioned approach towards full system memory encryption takes a probabilistic approach that reduces the risk of having sensitive content stored in the memory. This however leaves a door for the aforementioned software attacks that allow kernel to read the entire address space of application. Because memory coherence is maintained automatically by the processor, OS kernel could directly read out the private data in the working set, regardless they reside in DRAM, on-chip memory or caches. With CryptMe, this decrypted working set is protected within the processor boundary in the iRAM against the cold boot attack. The iRAM is further protected by the TrustZone memory separation against memory disclosure attacks due to misused OS functions.

7.2 TrustZone-Based Solutions

TrustZone is a system wide security extension on ARM processors. Due to its unique ability to provide isolated execution environment even when the software of the system is compromised, TrustZone has been widely adopted in both academia research project and commercial project [4,15,20,23,32,36]. CaSE [36] is a system closely related to CryptMe. In CaSE, sensitive workloads are encrypted and only decrypted during execution completely within the processor cache in ARM system to address the threat from physical memory disclosure. However, CaSE has limitation on the size of application binary. CryptMe utilizes the iRAM for storing sensitive data and extends its capacity by employing a sliding-window algorithm. Therefore, it can support unmodified binaries of arbitrary size. TrustShadow [15] resembles our work in that we both offload the execution of trusted applications to the secure world. However, TrustShadow focuses on defeating malicious OSes, while CryptMe focuses on defeating memory disclosure attacks.

8 Limitations and Future Work

Our design is not a full memory encryption solution which encrypts the whole address space of a process. Encrypted code is a compelling form of protection to thwart reverse-engineering of proprietary software. Although the current version of CryptMe does not protect the confidentiality of program code, it is possible to extend it to encrypt code segment as well. However, we anticipate that new issues will arises. For example, how to handle shared libraries with non-SenProcesses is challenging. Moreover, it will inevitably introduce overhead due to increased working set.

We observed noticeable overhead for micro-benchmarks such as the memory latency test shown in Fig. 5. The overhead in the CryptMe mainly originates from page swapping as is shown in Table 5. In the future, we plan to improve CryptMe through the following two aspects. First, we will seek a better way to adjust the size of sliding window for individual SenProcesses. The provided customization allows for personalized configuration to maximum the usage of the valuable iRAM resource. Second, within a given sliding window, we plan to find a smarter page replacement algorithm to minimize the occurrence of page swapping.

9 Conclusions

In this paper, we present CryptMe, a practical ME solution for the ARM-based IoT devices. CryptMe supports unmodified program working on encrypted memory, mitigating the threats caused by memory leakages. Sensitive data is only decrypted in the `iRAM` of the SoC to protect against physical memory disclosure attacks. The trusted process is offloaded into an isolated execution domain with TrustZone. Therefore, our solution can also defeat software memory disclosure attacks from other processes or even the OS. We have implemented a CryptMe prototype on a real ARM SoC board. Experiment results show that CryptMe effectively defeats a wide range of memory disclosure attacks. Furthermore, CryptMe introduces moderate overhead for computation intensive programs, and negligible overhead for programs with small memory footprints. CryptMe enables ME for *unmodified* programs on the widely deployed ARM platforms. With small trade-off on the performance, CryptMe provides its users with unprecedented protection for private user data.

Acknowledgement. We thank the anonymous reviewers for their valuable comments. This work was supported by NSF CNS-1422594, NSF CNS-1505664, NSF SBE-1422215, and ARO W911NF-13-1-0421 (MURI). Neng Gao and Ji Xiang were partially supported by NSFC (No. U163620068). Jingqiang Lin was partially supported by NSFC (No. 61772518).

References

1. Apache Software Foundation: Apache HTTP server benchmarking tool (2017). http://httpd.apache.org/docs/2.4/programs/ab.html
2. ARM Holdings: mbed TLS (2017). https://tls.mbed.org/
3. ARM Ltd.: Arm cortex-a57 mpcore processor technical reference manual (2013)
4. Azab, A.M., et al.: Hypervision across worlds: real-time kernel protection from the arm trustzone secure world. In: ACM CCS (2014)
5. Becher, M., Dornseif, M., Klein, C.: Firewire: all your memory are belong to us. In: 6th Annual CanSecWest Conference (2005)
6. Chan, E.M., Carlyle, J.C., David, F.M., Farivar, R., Campbell, R.H.: Bootjacker: compromising computers using forced restarts. In: 15th ACM CCS. ACM (2008)
7. Chow, J., et al.: Understanding data lifetime via whole system simulation. In: USENIX SEC (2004)
8. Colp, P., et al.: Protecting data on smartphones and tablets from memory attacks. In: ASPLOS 2015. ACM (2015)
9. CVE Details: The Ultimate Security Vulnerability Datasource (2018). https://www.cvedetails.com/vendor/33/Linux.html. Accessed 29 Mar 2018
10. FuturePlus System: DDR2 800 bus analysis probe (2006). http://www.futureplus.com/download/datasheet/fs2334_ds.pdf
11. Garcia-Morchon, O., Kumar, S., Struik, R., Keoh, S., Hummen, R.: Security considerations in the IP-based internet of things (2013)
12. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M.: Data lifetime is a systems problem. In: 11th ACM SIGOPS European Workshop (2004)
13. Götzfried, J., Müller, T., Drescher, G., Nürnberger, S., Backes, M.: RamCrypt: kernel-based address space encryption for user-mode processes. In: 11th ACM Asia CCS. ACM (2016)
14. Götzfried, J., et al.: Hypercrypt: hypervisor-based encryption of kernel and user space. In: ARES 2016 (2016)
15. Guan, L., et al.: Trustshadow: secure execution of unmodified applications with arm trustzone. In: ACM MobiSys (2017)
16. Halderman, J.A., et al.: Lest we remember: cold boot attacks on encryption keys. In: USENIX SEC (2008)
17. Harrison, K., Xu, S.: Protecting cryptographic keys from memory disclosure attacks. In: IEEE/IFIP DSN (2007)
18. Henson, M., Taylor, S.: Beyond full disk encryption: protection on security-enhanced commodity processors. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 307–321. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38980-1_19
19. Henson, M., Taylor, S.: Memory encryption: a survey of existing techniques. ACM CSUR (2014)
20. Jang, J., Kong, S., Kim, M., Kim, D., Kang, B.B.: Secret: secure channel between rich execution environment and trusted execution environment. In: NDSS 2015 (2015)
21. Kleissner, P.: Hibernation file attack (2010)
22. Kolontsov, V.: Solaris (and others) ftpd core dump bug (1996). http://insecure.org/sploits/ftpd.pasv.html
23. Li, W., Li, H., Chen, H., Xia, Y.: Adattester: secure online mobile advertisement attestation using trustzone. In: ACM MobiSys (2015)

24. Lie, D.: Architectural support for copy and tamper resistant software. ACM SIGPLAN Not. **35**, 168–177 (2000)
25. McVoy, L., Staelin, C.: Lmbench: portable tools for performance analysis. In: USENIX ATC (1996)
26. Müller, T., Spreitzenbarth, M., Freiling, F.: FROST: forensic recovery of scrambled telephones. In: 11th ACNS (2013)
27. National Vulnerability Database: CVE-2011-2707 (2011). http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2011-2707
28. National Vulnerability Database: CVE-2005-1264 (2015). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1264
29. Peterson, P.A.: Cryptkeeper: improving security with encrypted RAM. In: IEEE HST (2010)
30. Provos, N.: Encrypting virtual memory. In: USENIX SEC (2000)
31. Reese, W.: Nginx: the high-performance web server and reverse proxy (2008). https://nginx.org/
32. Santos, N., Raj, H., Saroiu, S., Wolman, A.: Using ARM trustzone to build a trusted language runtime for mobile applications. In: ASPLOS 2014. ACM (2014)
33. Stewin, P., Bystrov, I.: Understanding DMA malware. In: Flegel, U., Markatos, E., Robertson, W. (eds.) DIMVA 2012. LNCS, vol. 7591, pp. 21–41. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37300-8_2
34. Suiche, M.: Windows hibernation file for fun ‘n’ profit. Black-Hat (2008)
35. Wilson, P., et al.: Implementing embedded security on dual-virtual-CPU systems. IEEE Des. Test Comput. (2007)
36. Zhang, N., Sun, K., Lou, W., Hou, Y.T.: Case: cache-assisted secure execution on arm processors. In: IEEE S&P (2016)