Secure In-Cache Execution

Yue Chen, Mustakimur Khandaker, and Zhi Wang✉

Florida State University, Tallahassee, FL, US 32306
{ychen, khandake, zwang}@cs.fsu.edu

Abstract. A cold boot attack is a powerful physical attack that can dump the memory of a computer system and extract sensitive data from it. Previous defenses focus on storing cryptographic keys off the memory in the limited storage "borrowed" from hardware chips. In this paper, we propose EncExec, a practical and effective defense against cold boot attacks. EncExec has two key techniques: spatial cache reservation and secure in-cache execution. The former overcomes the challenge that x86 processors lack a fine-grained cache control by reserving a small block of the CPU's level-3 cache exclusively for use by EncExec; the latter leverages the reserved cache to enable split views of the protected data: the data stored in the physical memory is always encrypted, and the plaintext view of the data is strictly confined to the reserved cache. Consequently, a cold boot attack can only obtain the encrypted form of the data. We have built a prototype of EncExec for the FreeBSD system. The evaluation demonstrates that EncExec is a practical and effective defense against cold boot attacks.

1 Introduction

A cold boot attack is a powerful physical attack that can extract sensitive data from the physical memory[1] of a computer system. It exploits the fact that, contrary to the common belief, memory chips may retain their contents for seconds after the power is lost and considerably longer at a low temperature [14, 27]. An attacker can dump the memory of a victim computer by freezing and transplanting its memory units to a computer under his control or rebooting it to a malicious operating system (OS). Sensitive data can then be extracted from the dumped memory [14]. Lots of sensitive data sit in the memory for a long time [7]. For example, whole-disk encryption protects the document at rest in case the computer is lost or stolen. However, the disk encryption key (or its derived sub-keys) often sits in the memory in plaintext and thus vulnerable to the cold boot attack. Cold boot attacks have also been demonstrated against mobile devices, even though their memory units are soldered onto the motherboard [23, 28], by freezing and rebooting them to the recovery mode . The attacker then uses a tool to extract sensitive data from the phone, including passwords, contacts, photos, and emails. Cold boot attacks have become a major security and privacy concern.

A few defenses have been proposed to address cold boot attacks on the x86 [12, 21, 22, 26] and ARM [10] platforms. In principle, they re-purpose existing hardware features to keep cryptographic keys off the memory. For example, AESSE [21], TRESOR [22], LoopAmnesia [26], and ARMORED [10] store a single AES key in SSE registers,

[1] For brevity, we refer to the physical memory as the memory and the CPU cache as the cache.

debug registers, performance counters, and NEON registers, respectively. By doing so, the key will never leave the CPU and consequently not be contained in the memory dump. However, the amount of storage provided by these "borrowed" registers is very limited. It is often too small for cryptographic algorithms that use longer keys (e.g., RSA). They also interfere with normal operations of these registers. From another perspective, Copker [12] temporarily disables caching and uses the cache-as-RAM technology [9] to implement RSA. However, Copker severely degrades the system performance when it is active because caching has to be completely disabled. On recent Intel processors with a shared Level-3 (L3) cache, Copker has to disable caching on *all* the cores. Moreover, these systems focus solely on securing cryptographic algorithms while completely ignoring other sensitive data in the process (one reason is that they do not have large enough secure storage for them.) Sensitive data, such as user accounts and passwords, can be scattered in the process address space as the memory is allocated, copied, and freed [7]. This calls for a stronger protection against cold boot attacks that can protect not only cryptographic keys but also sensitive data.

In this paper, we propose EncExec, a system that can securely execute a whole program, or a part of it, in the cache. Data protected by EncExec have split views in the memory and the (reserved) cache: data stored in the memory are always encrypted; they are decrypted into the cache only when accessed. EncExec guarantees that the decrypted data will *never* be evicted to the memory. As such, the reserved cache is desynchronized from the memory. Even though the data are encrypted in the memory, the CPU can still access the unencrypted data from the cache because the cache precedes the memory. Consequently, the memory dump contains just the encrypted view of the protected data. Their unencrypted view only exists in the cache and will be lost when the power is reset or the system is rebooted. To enable split views of the protected data, EncExec relies on two key techniques, spatial cache reservation and secure in-cache execution. The former reserves a small block of the cache by carefully managing the system's physical memory allocation. A key challenge here is the lack of direct control of the cache in the x86 architecture – there are instructions to enable/disable the cache and to invalidate the whole cache or a cache line, but there is no fine-grained control over how data is cached and evicted by various levels of caches. Without precise control of cache replacement, the unencrypted data in the cache could be accidentally leaked to the memory. To address that, we observe that x86 processors use the n-way set-associative cache organization. EncExec thus can reserve a small block of the cache by reserving all the physical memory cached by it. Additionally, the CPU will not spontaneously evict a cache line unless there are cache conflicts. EncExec thus can prevent the unencrypted data from being evicted to the memory by avoiding conflicts for the reserved cache. EncExec's second technique utilizes the reserved cache to protect sensitive data by desynchronizing the cache and the memory.

EncExec can be used in two modes. In the first mode, a process is given a block of the secure memory for storing its critical data. The process can decide which data to protect. From the process' point of view, this block of memory can be used just like the regular memory. In the second mode, EncExec uses the reserved cache to protect the whole data of the process. Specifically, it uses the reserved cache as a window over the process' data, similar to demand paging. The data in the window are decrypted in

the cache and remain in the cache until they are replaced by EncExec. The data out of the window only exist in the memory and stay encrypted. Note that window-based encrypted execution alone is not secure because the (unencrypted) window often contain critical data due to program locality. For example, a web server's private key most likely is in the window because it is constantly being used to encrypt and decrypt web traffic. Without strict cache control provided by EncExec's first technique, the unencrypted data can be evicted to the memory and obtained by an attacker. Between these two modes, the first one is more practical because a process has the best knowledge of its data and the reserved cache is still relatively small for large programs. The first mode can support more processes simultaneously. However, it does require some changes to the program.

We have built a prototype of EncExec for the FreeBSD 10.2 operating system. Our prototyping experience shows that EncExec can be easily integrated into the kernel and provide an effective defense against cold boot attacks. The performance overhead is very minor for the first mode, while the overhead for the second mode as expected depends mostly on the process' program locality.

2 System Design

2.1 Design Goals and Assumptions

EncExec aims at protecting a process' sensitive data against cold-boot attacks. Specifically, it reserves a small block of the (L3) cache and uses the reserved cache to securely execute the whole process or a part of it in the cache. We have the following design goals for EncExec:

- *Data secrecy*: the plaintext view of the protected data should only exist in the cache. It must be encrypted before being evicted to the memory. The key to encrypt the data must be protected from cold boot attacks as well.
- *Data quantity requirement*: most early defenses can only secure small cryptographic keys. A practical solution should support cryptographic algorithms such as RSA that use large keys.
- *Performance isolation*: the cache is critical to the overall system performance. EncExec reserves a small portion of the L3 cache for its use. It should not incur large performance overhead for other processes whether EncExec is active or not; i.e., the performance impact of EncExec is isolated from concurrent processes.
- *Application transparency*: when operating in the whole-data protection mode, EncExec should be transparent to the protected process. An unmodified user program should be able to run under EncExec just like on a normal OS (but slower).

Threat Model: the attacker is assumed to have physical access to the victim's device. He can launch a cold-boot attack either by transplanting the (frozen) memory units to a computer under his control [14] or by rebooting it to a tiny malicious OS [23, 28]. We assume that the attacker does not have malware, such as a kernel rootkit, installed on the victim's device, otherwise he could simply obtain the memory through the malware without resorting to cold-boot attacks. This threat model covers the common scenarios where cold-boot attacks may be attempted. For example, many business laptops lost in public places have encrypted hard disks and are protected by screen locks.

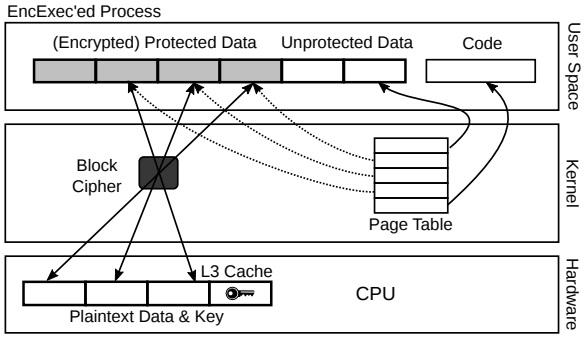

Fig. 1: Overall architecture of EncExec. Three pages of the protected data is in the window.

Since the attacker has physical control over the device, he could launch other physical attacks. For example, external expansion buses like FireWire may be exploited to directly access the physical memory via DMA. Some devices have enabled debug ports (e.g., the JTAG port on a mobile phone). The attacker can attach a debugger to these ports and fully control the system. More exotic attacks, such as monitoring or injecting data on the buses, often require sophisticated equipment and aplenty financial support. In this paper, we consider these attacks out of the scope and assume they are prevented by other defenses, such as using IOMMU to prevent DMA attacks and disabling debug ports.

A process may have close interaction with its external environment. Sensitive data could leak to the environment. For example, a word processor often stores parsed documents in temporary files. This problem has been addressed by a number of previous systems [24]. In this paper, we assume the data transferred out of the process maintain their secrecy by, say, encrypting the file system and network communications. Of course, the keys for encryption need to be protected (by EncExec).

2.2 Design Overview

Figure 1 shows the overall architecture of EncExec. The user space of a process consists of code and data sections. EncExec focuses on protecting the process' data against cold boot attacks but leaves the code as is. This is based on the assumption that the data more likely contain sensitive information that needs protection, while the code is often publicly available and does not carry private user information. Nevertheless, EncExec can also be applied to protect the code if needed. In Fig. 1, the protected data remain encrypted in the memory all the time, the decrypted data are stored in the part of the L3 cache reserved by EncExec. EncExec uses the L3 cache to minimize its performance impact because recent Intel processors have large, unified, inclusive L3 caches [2]. Moreover, the L3 cache is *physically indexed* and *physically tagged*. Physical addresses thus solely determine the allocation of this cache. To enable the split views of the data, EncExec uses the reserved cache as a (discontinuous) window over the protected data. The data in

[2] An unified cache stores both code and data. An inclusive L3 cache includes all the data cached by the L1 and L2 caches.

the window is decrypted and fully contained within the cache. Since the cache precedes the memory, the CPU directly accesses the decrypted data in the cache when it tries to access the data in the window. The data out of the window remains encrypted in the memory and unaccessible to the CPU. EncExec extends the kernel's virtual memory management (VMM) to strictly control the process' data access so that no plaintext data will be evicted to the memory due to cache conflicts. Specifically, only the protected data in the window (as well as the code and unprotected data) are mapped in the process' address space. If more protected data are used than the window size, EncExec selects a page in the window for replacement, similar to demand paging in the OS. Because a page table can only map memory in pages, the reserved cache must be page-aligned, and its size is a multiple of the page size. We use the hardware-accelerated AES (AES-NI [16]) in the counter mode for encryption. Both the key and the initial vector are randomly generated, and the key and sub-keys are securely stored in the reserved cache to protect them from cold boot attacks.

This architecture can support both modes of EncExec. In the first mode, EncExec provides the process with a block of secure memory. Any data stored in this memory is guaranteed to be protected from cold boot attacks. The process can decide when and how to use this memory. As such, the program needs to be (slightly) modified. For example, we can modify a cryptographic library so that its stack and session data are allocated from the secure memory. In the second mode, EncExec protects the complete data of a process. This mode is transparent to the protected process; no changes to the program are necessary. Because we use demand-paging to manage the reserved cache, the amount of the protected data can be larger than the size of the reserved cache for both modes, similar to how virtual memory can be larger than the physical memory. In the rest of this section, we describe in detail the design of EncExec. EncExec has two key techniques: spatial cache reservation reserves a small, continuous block of the L3 cache for exclusive use by EncExec, secure in-cache execution leverages the reserved cache to protect the process data.

2.3 Spatial Cache Reservation

EncExec's first technique reserves a small part of the L3 cache for its use. This is a challenging task on the x86 architecture because x86 transparently manages cache assignment and replacement. It does not provide explicit and fine-grained control of the cache. A process has no direct control over how its data are cached, and the CPU decides transparently which cached line [3] to be replaced when there is a cache conflict. To replace a cache line, the CPU first evicts the old contents back to the memory and then loads the new contents from the memory. EncExec needs to precisely control how the protected data are cached and how the cache is replaced to avoid conflicts in the reserved cache. Without this control, the CPU can evict some of the reserved cache to the memory, leaking the unencrypted data to the physical memory. To address that, EncExec enforces the following two rules:

[3] A cache line is the unit of data transfer between the cache and the memory. Recent x86 processors use a cache line of 64 bytes.

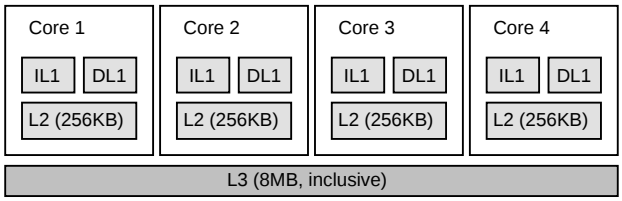

Fig. 2: Intel Core i7 cache architecture

– *Rule I*, the protected data are only cached by the reserved cache, and no other memory is cached by the reserved cache. Consequently, neither the kernel itself nor other processes can cause the reserved cache to be evicted.
– *Rule 2*, the amount of the *accessible* (decrypted) protected data must be less than the size of the reserved cache. They thus always fit in the reserved cache. Consequently, the protected data themselves cannot cause the reserved cache to be evicted.

These two rules prevent conflicts in the reserved cache caused by other processes and by the protected data themselves, respectively. With these two rules, EncExec can guarantee that the decrypted data remain in the CPU cache, unobtainable by cold boot attacks.

EncExec enforces these two rules by leveraging the cache architecture and the replacement algorithm of x86 processors. Modern x86 processors often have a large shared L3 cache. Fig. 2 shows the cache architecture of an Intel Core-i7 4790 processor. There are three levels of caches. Each CPU core has dedicated L1 and L2 caches, but all the four cores share a single large L3 cache. The L1 cache is split into an instruction cache (IL1) and a data cache (DL1), each 32KB in size. The L2 and L3 caches are unified in that they cache both code and data. L1 and L2 caches are relatively small in size (64KB and 256KB, respectively), but the L3 cache is capacious at 8MB. Even though the L1 and L2 caches are small, they are more important to the overall system performance because they are faster and closer to CPU cores. It is thus impractical to reserve any part of the L1 or L2 cache, especially because EncExec has to reserve the cache in pages (4KB at least). Another important feature of the L3 cache for EncExec is inclusivity. An inclusive L3 cache is guaranteed to contain all the data cached by the L1 and L2 caches. The CPU will not bypass the L3 cache when it accesses the memory. Without inclusivity, the CPU could evict the unencrypted data to the memory directly from the L1 or L2 cache and load the encrypted data directly from the memory to the L1 or L2 cache. The former leaks the unencrypted data to the memory, while the latter causes the program to malfunction. Recent Intel processors have large, unified, inclusive L3 caches. However, old processors like Pentium 4 have non-inclusive L2 caches (they do not have an on-chip L3 cache.) and thus cannot be used by EncExec. In addition, we assume the cache is set to the write-back mode instead of the write-through mode. This is because in the write-through mode the CPU keeps the cache and the memory in sync by writing any updates to the cached to the memory as well. Most OSes use the write-back mode for the main memory due to its better performance.

EncExec takes control of all the physical memory cached by the reserved cache so that no other processes can use that memory and cause eviction of the reserved cache (rule 1). The actual memory EncExec has to control is decided by the CPU's cache

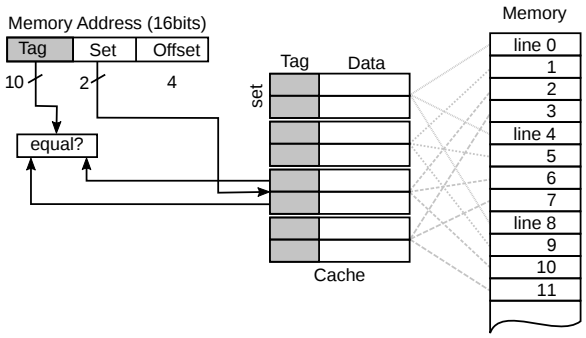

Fig. 3: 2-way set-associative cache, 8 cache lines in 4 sets. Each cache line is 16 bytes.

organization. Specifically, the memory and the cache are divided into equal-sized cache lines (64 bytes). The memory in a line is cached or evicted as a whole. Intel processors use the popular *n*-way set-associative algorithm to manage the cache [15]. Fig. 3 shows a simple 2-way set-associative cache with a cache line of 16 bytes to illustrate the concept. This cache is organized into 8 cache lines, and each two consecutive lines are grouped into a set. This cache thus has 8 cache lines in 4 sets. Meanwhile, the memory address (16 bits) is divided into three fields: the `offset` field (4 bits) specifies the offset into a cache line. This field is ignored by the cache since the memory is cached in lines; the `set` field (2 bits) is an index into the cache sets. A line of the memory can be cached by either line in the indexed set; the last field, `tag`, uniquely identifies the line of the physical memory stored in a cache line. During the cache fill, the CPU selects one line of the indexed set (evict it first if it is used) and loads the new memory line into it. The tag is stored along with the data in the cache line. During the cache lookup, the CPU compares the tag of the address to the two tags in the indexed set simultaneously. If there is a match, the address is cached (a cache hit); otherwise, a cache miss has happened. The CPU then fills one of the lines with the data. Note that all the addresses here are physical addresses as the L3 cache is physically indexed and physically tagged.

The L3 cache of Intel Core-i7 4790 is a 16-way set-associative cache with a cache line size of 64 bytes [15]. Therefore, the `offset` field has 6 bits to address each of the 64 bytes in a cache line. The width of the `set` field is decided by three factors: the cache size, the cache line size, and the associativity. This processor has an 8MB L3 cache. The `set` field thus has 13 bits ($\frac{8M}{64\times16} = 8192 = 2^{13}$); i.e., there are 8, 192 sets. The `tag` field consists of all the leftover bits. If the machine has 16GB (2^{34}) of physical memory (note the L3 cache is physically tagged), the *tag* field thus has 15 bits ($34 - 6 - 13 = 15$).

EncExec relies on the page table to control the use of the reserved memory (Section 2.4). A page table can only map page-sized and page-aligned memory. Therefore, EncExec has to reserve at least a whole page of the L3 cache. Even though this processor supports several page sizes (4KB, 2MB, and 1GB), we only reserve a smallest page of the cache (4KB, or 64 lines) to minimize the performance overhead. However, we have to reserve 64 cache sets instead of 64 cache lines because this cache uses 16-way set-associative and all the cache lines in the same set have to be reserved together (as

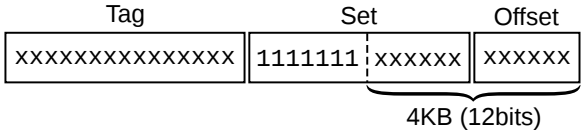

Fig. 4: Addresses that map to the reserved cache (bits marked with x can be either 0 or 1.)

the CPU may cache our data in any line of a set). The actual amount of the reserved cache accordingly is 64KB. These reserved cache sets must be continuous and the first set is page-aligned so that together they can cache a whole page of the physical memory. In our prototype, we reserve the cache sets from index 8,128 (`0x1FC0`) to index 8,191 (`0x1FFF`). Fig. 4 shows the format of memory addresses that are cached by these selected sets. EncExec needs to take control of all physical pages conforming to this format (rule 1), which total $\frac{1}{128}$ of the physical memory. For example, it needs to reserve `128MB` physical memory on a machine with `16GB` of RAM. As mandated by rule 2, EncExec cannot use more data than the reserved cache size in order to avoid cache conflicts. Therefore, we can use 16 pages (64KB) of the reserved 128MB memory at a time. Note that the amount of the protected data can be larger than 16 pages because we use demand paging to manage the reserved cache. Moreover, an attacker that controls an unprotected process (e.g., using JavaScript in a browser) cannot evict EncExec's reserved cache because the reserved physical memory is not mapped in that process's virtual address space (remember the L3 cache is physically indexed and physically tagged.)

2.4 Secure in-Cache Execution

EncExec's second technique, secure in-cache execution, splits the views of the protected data between the memory and the reserved cache: the data remain encrypted in the memory, and their plaintext view only exists in the cache. In other words, we need to desynchronize the memory and the cache. There are three requirements for this to happen: *first*, the cache must be configured to use the write-back mode so that data modified in the cache will not be written through to the memory; *second*, the L3 cache is inclusive of the L1 and L2 caches so that the CPU always accesses the memory through the L3 cache; *third*, there are no conflicts in the reserved cache so that the CPU will not evict any reserved cache line. The first two requirements are guaranteed by the hardware and the existing kernels. The third requirement is fulfilled by EncExec's second technique.

EncExec's first technique takes control of all the physical pages that may be cached by the reserved cache. As long as we use no more protected data than the size of the reserved cache, they can fit in the reserved cache without conflicts, and any changes to these data, including decryption, stay within the cache. To continue the previous example, we select 16 pages out of the reserved 128MB physical memory and use these pages for securing the protected data. We call these pages plaintext pages. In order to desynchronize the cache and the memory, we only need to copy the encrypted data to a plaintext page and decrypt them there. The decrypted data remain in the cache since there are no cache conflicts. However, we often need to protect more data than that can

fit in plaintext pages. To address that, EncExec's second technique leverages demand paging to protect a large amount of data.

In demand paging, a part of the process can be temporarily swapped out to the backing store (e.g., a swap partition) and be brought into the memory on-demand later [25]. For EncExec, the original memory of the protected data serves as the swap for plaintext pages. The data are brought into and evicted from plaintext pages when necessary. The page table is used to control the process' access to the data. When the process tries to access the unmapped data (marked as non-present in the page table), the hardware delivers a page fault exception to the kernel. EncExec hooks into the kernel's page fault handler and checks whether the page fault is caused by the unmapped protected data. If so, it tries to allocate a plaintext page for the faulting page. If none of the plaintext pages are available, EncExec selects one for replacement. Specifically, it first encrypts the plaintext page and copies it back into its original page (note the original page is a non-reserved page). EncExec then decrypts the faulting page into this plaintext page and updates the page table if necessary. To initiate the protection, EncExec encrypts all the protected data, flushes their cache, and unmaps them from the process' address space. As such, EncExec can completely moderate access to the protected data. By integrating EncExec into the kernel's virtual memory management, we can leverage its sophisticated page replacement algorithm (e.g., the LRU algorithm) to select plaintext pages for replacement. Note that the second technique alone is not secure because plaintext pages often contain (most recently used) sensitive data due to program locality. Without the first technique, there is no guarantee that plaintext pages will not be evicted to the memory and become vulnerable to cold boot attacks. It is thus necessary for both techniques to work together.

EncExec needs to frequently encrypt and decrypt the protected data. The cryptographic algorithm thus plays an important role in the overall performance of the protected process. Recent CPUs have built-in hardware support to speed up popular cryptographic algorithms. For example, most Intel CPUs now feature hardware acceleration for AES, a popular block cipher, through the AES-NI extension [16]. EncExec uses hardware-accelerated ciphers when available. Our prototype uses AES-128 in the counter mode. Therefore, each block of the protected data can be encrypted/decrypted independently. To protect the (randomly generated) key from cold boot attacks, we dedicate one plaintext page to store the key, its derived sub-keys, and other important intermediate data. Initial vectors are stored for each page of the data. It is not necessary to protect the secrecy of initial vectors, but they should never be reused.

3 Implementation

We have implemented a prototype of EncExec based on the FreeBSD operating system (64-bit, version 10.2) [1]. FreeBSD's virtual memory management has an interesting design that enables multiple choices for implementing EncExec. Our prototype is based on the Intel Core-i7 4790 CPU with 16GB of memory. Other CPUs with a similar cache architecture can be used by EncExec as well. For example, the Xeon E5-2650 processor has a 20MB shared, inclusive L3 cache organized as 20-way set-associative. In the rest of this section, we describe our prototype in detail.

3.1 Spatial Cache Reservation

EncExec reserves a block of the L3 cache by owning all the physical pages cached by it, i.e., physical pages whose addresses are all 1's from bit 12 to bit 18 (Fig. 4). In other words, EncExec reserves one physical page every 128 pages [4]. EncExec can only use 16 of these pages as plaintext pages to protect data so that these pages can be completely contained in the reserved cache.

Modern operating systems have a sophisticated, multi-layer memory manage system to fulfill many needs of the kernel and the user space. For example, physical memory is often allocated in pages by the buddy memory allocator, and kernel objects are managed by the slab memory allocator to reduce fragmentation [1]. The slab allocator obtains its memory from the physical page allocator, forming a layered structure. Given this complexity, EncExec reserves its physical pages early in the booting process when the kernel is still running in the single processor mode. Memory allocated after enabling the multi-processor support is harder to reclaim – an allocated page may contain kernel objects accessed concurrently by several processors. Simply reclaiming it will lead to race conditions or inconsistent data.

When FreeBSD boots, the boot loader loads the kernel image into a continuous block of the physical memory, starts the kernel, and passes it the layout of the physical memory discovered by the BIOS (through `e820` memory mappings). The kernel image is large enough to contain several reserved physical pages that need to be reclaimed. The kernel uses several ad-hoc boot-time memory allocators to allocate memory for the booting process (e.g., `allocpages` in `sys/amd64/amd64/pmap.c`). We modify these allocators to skip the reserved pages. If a large block of memory (i.e., more than 127 pages) is allocated, we save the location and length of the allocated memory in order to fix it later. A typical example is the array of `vm_page` structures. There is one `vm_page` structure for each physical page. This structure thus could be really large.

By now, the kernel still runs on the simple boot page table. The kernel then replaces it with a new, more complete page table (`create_pagetables` in `sys/amd64/amd64/pmap.c`). x86-64 has a huge virtual address space. This allows the kernel to use it in ways that are not possible in 32-bit systems. For example, the kernel has a dedicated 4TB area that directly maps all the physical memory, including the reserved pages. This is called the direct map. Accordingly, the kernel can access any physical memory by adding the offset of this area to the physical address (`PHYS_TO_DMAP(pa)`). It is not necessary to unmap plaintext pages from the direct map because EncExec needs to directly access them (e.g., to decrypt a page). Another area in this new page table maps in the kernel. As mentioned earlier, the kernel is large enough to contain several reserved pages. If we find such a page when creating the page table, we allocate a non-reserved page, copy the contents from the original page, and map this page in the page table [5]. This essentially replaces all the reserved pages in the kernel map with non-reserved pages. The kernel then switches to the new page table and continues the booting process. To make sure that no reserved pages exist in this new page table (except the DMAP area), we write a

[4] This number is decided by the CPU's cache architecture. For example, EncExec reserves one physical page every 512 pages on the aforementioned Xeon E5-2650 CPU.

[5] The kernel uses 2MB large pages to map its data. We break them down into 4KB pages first.

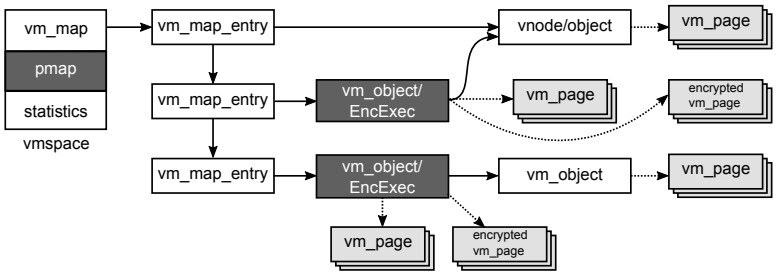

Fig. 5: Virtual memory structures for a process. Dark rectangles mark the possible placements for EncExec.

small kernel function to scan the new page table for reserved pages. No reserved page was found in the new page table.

Eventually, the kernel adds the left-over physical pages to the run-time page allocator (`vm_page_startup` in `sys/vm/vm_page.c`). We hook into this function and reserve pages as they are being added to the allocator. We also hook into the kernel's whole-cache flushing function (`wbinvd`). By doing so, we can encrypt the plaintext pages first to prevent the unencrypted data from being leaked to the memory. The CPU may temporarily power down a part of its caches in some power management states. In our implementation, we disable the CPU power management when EncExec is active and re-enable it otherwise.

3.2 Secure in-Cache Execution

With EncExec, the protected data remain encrypted in the memory; they are loaded and decrypted into the reserved cache on demand. This essentially adds another layer to the kernel's demand paging: the memory serves as the backing store for plaintext pages, while the swap partition services as the backing store for the memory. EncExec manipulates the page table to desynchronize the cache and the plaintext pages. FreeBSD's virtual memory management has a unique design that provides two alternative choices in the placement of EncExec.

Figure 5 shows the kernel data structures that manage the process address space in FreeBSD. The top level structure, `vmspace`, represents the whole address space. It is a container for other related structures. Two notable such structures are `vm_map` and `pmap`. FreeBSD separates the address space layout (`vm_map`) from the page table management (`pmap`). `vm_map` describes the process' virtual address space layout using a list of `vm_map_entry` structures. Each `vm_map_entry` specifies a continuous block of the process' address space, including its start and end addresses and its permissions (read/write/execute). Each `vm_map_entry` is backed by a chain of `vm_objects`. A `vm_object` describes the origin of the data for this entry and the backing store to swap in and out the data. There are three types of `vm_objects`. Named `vm_objects` represent files. Program sections like the code and data sections use named `vm_objects` because their initial contents are loaded from the program binary. Anonymous `vm_objects` represent sections that are zero-filled on the first use, such as uninitialized data sections

and heap sections. Shadow vm_objects hold a private copy of the locally modified pages (represented by vm_pages). Every vm_object has an associated pager interface that decides how to swap in and swap out the object's associated data. For example, anonymous vm_objects use the swap pager to store data in the swap partition. The pmap structure consists of architecture-specific data and functions to manage the process' page table. Every CPU architecture defines its own pmap structure but implements the common pmap API. As such, other kernel modules do not need to be concerned with the details of page tables. Pmap can decide when and how to map a page. For example, it can unmap a page from the process' address space as long as the page is not pinned by the upper vm layers.

This design enables two feasible ways to implement EncExec in the FreeBSD kernel: it can either be implemented as a shadow object or in the pmap module. We chose the latter because it is simpler and more likely to be applicable to other OSes (e.g., Linux). Specifically, a vm_map_entry can be backed by a chain of vm_objects. Objects ahead in the chain precede over these later in the chain. When a page fault happens, the page fault handler searches for the faulting page along the chain of vm_objects. It returns the first located page without checking the rest of the chain. This chain of objects is essential to many features of FreeBSD's virtual memory design, such as copy-on-write where the kernel creates a shadow object of the original one for both the parent and the child and marks the original object read-only. If either process tries to modify a shared page, the kernel makes a copy of the page and gives it to the corresponding shadow object. This new copy overshadows the original shared page. EncExec can be similarly implemented as a shadow object by using plaintext pages to store the data and the original (non-reserved) memory as the backing store. However, this design introduces additional complexity to the kernel's already tangled virtual memory system [20]. For example, EncExec's shadow object should always be the first object in the chain, otherwise plaintext pages could be copied to vm_objects earlier in the chain and leaked to the memory. EncExec thus has to monitor any changes to the object chain. If a new object is inserted before EncExec's shadow object, EncExec must move its object to the head of the chain. Reordering objects is not supported by FreeBSD. Additionally, it is hard to apply this design to other kernels that do not have a similar structure (e.g., Linux).

The pmap module also has the needed support for EncExec: by design, pmap is allowed to unmap a page from the process' address space as long as the page is not pinned. The page fault handler will ask pmap to remap that page if it is later accessed. Moreover, pmap maintains a reverse mapping for each physical page, which keeps track of all the processes and virtual addresses a physical page is mapped. EncExec uses reverse mapping to completely disconnect a shared page from all the processes, otherwise some processes might incorrectly access the encrypted data. Pmap also tracks the page usage information for page replacement. EncExec can leverage this information to optimize its own page replacement.

EncExec first picks 15 reserved pages as the plaintext pages and unmaps all the protected data from the process' address space. If a page is shared by multiple processes, EncExec removes it from other processes as well. EncExec then returns to the user space to resume the process. When the process accesses its protected data, a page fault will be triggered. The page fault handler searches its data structures and asks pmap to map in the

data (`pmap_map` in `pmap.c`). EncExec intercepts this request, allocates a plaintext page, decrypts the target page into it, and maps it into the process. At any time, no more than 15 plaintext pages can be used by the protected process. If more are needed, EncExec will pick a plaintext page in use for replacement. In addition, the FreeBSD kernel might proactively map-in (also called pre-fault) a page, expecting the process to access it in the near future. No page fault will be triggered when the process accesses this page later. In our prototype, we disable pre-faulting for the protected data sections to save the limited plaintext pages. The process may also perform file I/O with the protected memory. For correctness, we temporarily restore the affected pages and unmap them from the process' address space. When these pages are accessed by the user space again, page faults will be triggered. This signals the end of the file I/O operation. We then re-enable the protection for these pages.

4 Evaluation

In this section, we evaluate the security and performance of our EncExec prototype. All the experiments were conducted on a desktop with a 3.6 GHz Intel Core i7-4790 CPU and 16GB of memory. The system runs 64-bit FreeBSD for x86-64, version 10.2. To test its performance, we run various benchmarks in the FreeBSD ports and the benchmarks of mbed TLS [2]. Mbed TLS, formerly known as PolarSSL, is a popular, portable, open-source TLS library developed by ARM.

4.1 Validation

We first validate that we can actually desynchronize the cache and plaintext pages. Theoretically, updates to the plaintext pages should be confined to the reserved cache because neither the kernel nor other processes can cause cache conflicts with EncExec, and we never use more plaintext pages than the size of the reserved cache. Consequently, the CPU should not evict the cached plaintext pages. However, x86 does not provide instructions to directly query the cache line status. To address that, we first validate that none of the reserved pages are used by the kernel or unprotected processes. Specifically, we write a simple kernel function that scans a page table for reserved pages. We apply this function to all the active page tables in the system. No reserved pages are found to be mapped in these page tables, except plaintext pages in the kernel's direct map area. As mentioned before, we use direct map to encrypt/decrypt plaintext pages. We also unmap plaintext pages from the direct map when EncExec is not in use. If the kernel accesses any of these pages, a page fault will be triggered and the kernel will crash itself. None of these happen during our experiment. Moreover, we conduct an experiment to validate de-synchronization of the cache and the plaintext pages. Specifically, we write all zeros to a plaintext page and then execute the `wbinvd` instruction, which writes back the modified cache lines and invalidates the internal caches. Now, the cache and the page have been synchronized and the memory of this page is guaranteed to contain all zeros. Next, we modify the plaintext page. Any changes should remain in the cache. To check that, we discard the modified cache lines by executing the `invd` instruction and read the plaintext page again. The plaintext page should contain all zeros. This is indeed the case. It shows that the plaintext page and the cache have been desynchronized.

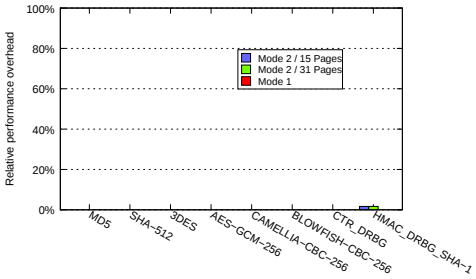

Fig. 6: Overhead of common cryptographic algorithms.

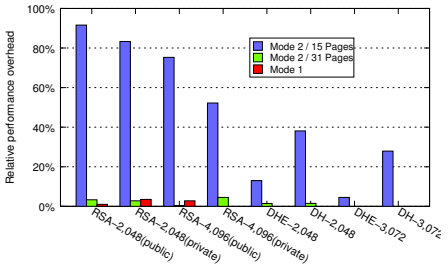

Fig. 7: Overhead of RSA and DH handshakes.

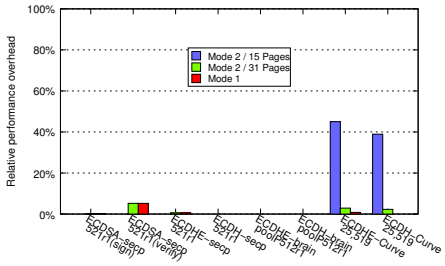

Fig. 8: Overhead of Elliptic Curve algorithms.

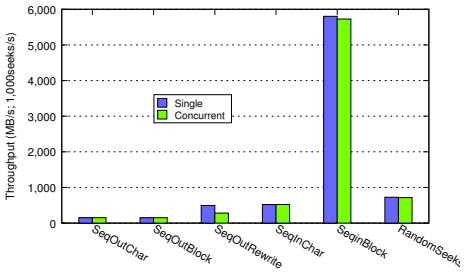

Fig. 9: Performance of Bonnie. The unit on Y-axis is MB/sec and thousandseeks/sec (for RandomSeeks only).

4.2 Performance Evaluation

EncExec uses the hardware-accelerated AES (AES-NI) to encrypt and decrypt data. Our measurements show that it takes about 3μs on average to encrypt/decrypt 4KB data using 128-bit AES algorithm. Therefore, there is an extra 3μs or so delay to load a data page and 6 μs if it's necessary to replace an existing page. This delay is the most significant source of EncExec's overhead, but it is hard to reduce this delay.

We use the official benchmarks from mbed TLS to measure the performance overhead of EncExec. These benchmarks consist of a wide range of cryptographic algorithms. The results are presented in Fig. 6, 7, and 8. The overhead is calculated relative to the baseline performance, i.e., the performance of mbed TLS on the original FreeBSD system. The "mode 1" bars give the overhead of mbed TLS protected by the first mode of EncExec. Specifically, we modified the source code of mbed TLS to allocate the session data and the stack from the secure memory. The "mode 2" bars give the overhead of mbed TLS protected by the second mode of EncExec, i.e., we protect all its data sections with EncExec. We experimented with both 15 and 31 plaintext pages. In the latter, we changed our prototype to double the reserved cache (32 pages, or 128KB). This set of experiments represents the most practical use cases of EncExec, given the limited size of the reserved cache.

For simple algorithms like SHA-512 and AES, EncExec incurs virtually no overhead (Fig. 6) because neither CPU nor the memory is a performance bottleneck. Earlier systems like TRESOR [22] have similar or even slightly better performance for AES. However, EncExec can support more complex algorithms, such as RSA and Diffie-Hellman, due to its larger secure storage. For those algorithms, mbed TLS in mode 1 and mode 2 / 31 pages only slightly lags behind the baseline (about 2% slower), but its performance under mode 2 / 15 pages is significantly slower than the baseline. For example, it can only achieve about 8.4% of the baseline performance for RSA-2048 public key encryption and 16.7% for RSA-2048 private key encryption. This can be explained with the working set model [25]. Clearly, the working set of these benchmarks is larger than 15 pages but less than (or around) 31 pages. With only 15 plaintext pages, thrashing is guaranteed, leading to poor performance. Mode 1 is not affected by the large working set because it only needs to protect the selected data, instead of all the data sections. Nevertheless, many real-world programs have very large working set. EncExec's second mode is thus more suitable for compact programs, such as an encryption/decryption service program.

We also measured the impact of EncExec on other concurrently running processes. Specifically, we run bonnie, a file system benchmark, twice, once alone and once while the mbed TLS benchmark is running (mode 2/ 15 pages). The results are shown in Fig. 9. These two runs have almost identical performance for most of the six tests except the third one: the concurrent run is about 43% slower. This overhead likely is not caused by EncExec but instead the result of the kernel's scheduling algorithm: both benchmarks have very low initial CPU usage. It is likely that they will be scheduled to run on the same CPU core. When bonnie is running its third test, the mbed TLS benchmark starts to run the RSA-related tests and uses more than 80% of the CPU. Temporarily, the mbed TLS benchmark preempts bonnie and degrades its performance. This is supported by the fact that bonnie uses 19.9% of the CPU time in the single run for this test, but it only receives 11.1% of the CPU time in the concurrent run. In the following tests, bonnie uses more than 100% of the CPU time and will be scheduled to a different CPU core than the mbed TLS benchmark. The performance of the mbed TLS benchmark remains mostly the same. To verify this hypothesis, we simultaneously run bonnie and the mbed TLS benchmark without EncExec. The similar results are observed a little bit earlier than the concurrent run with EncExec. This is because the mbed TLS benchmark runs faster this time. Overall, this result is not surprising: a process protected by EncExec and other processes cannot interfere with each other through the L3 cache, but they can still interact through the L1 and L2 caches if they are scheduled *to the same core*. Meanwhile, there is no interference through the cache if they are scheduled to different cores because each core has its own L1 and L2 caches. This strong performance isolation makes EncExec a more practical defense against cold boot attacks.

5 Discussion

In this section, we discuss some potential improvements to EncExec and related issues.

Impact on L1 and L2 Caches: EncExec controls all the physical pages cached by the reserved cache. This allows EncExec to precisely control the replacement of the

reserved cache. These pages are also cached by the L1 and L2 cache. This naturally raises the question of whether EncExec reserves some of the L1 and L2 cache, as an side effect. L1 and L2 caches are critical to the overall system performance as they are smaller, faster, and closer to CPU cores. Reserving even a small part of them could severely harm the system performance. Fortunately, EncExec does not reserve any of the L1 and L2 cache. This is because each L1 and L2 cache line can cache more physical lines than a L3 cache line does. For example, Intel Core i7-4790 has 256KB of L2 cache and 64KB of L1 cache (instruction + data). Its L2 cache uses the 8-way set-associative algorithm. Accordingly, the set field for the L2 cache is 9 bits ($\frac{256K}{64\times8} = 512 = 2^9$), and the tag field is 19 bits. Therefore, each L2 cache line caches 2^{19} lines of the physical memory, most of which are not reserved by EncExec. Therefore, EncExec does not reserve any of the L1 or L2 cache lines. Nevertheless, it changes access patterns of the L1 and L2 caches. Some L1 and L2 cache lines may see more activities and some less.

Thrashing Control: EncExec can protect either the selected sensitive data or all the data. In the latter case, thrashing could happen if the process' working set is larger than the reserved cache. To relieve that, we could reserve more cache and use processors with a larger L3 cache. For example, the Xeon E5-2670 processor has a 20MB shared L3 cache with 20-way set-associative. EncExec can use 40 plaintext pages (or 160 KB) if we reserve 8KB of the cache space. Recent Intel CPUs partition the L3 cache among their cores. Specifically, each core has its own slice of the CPU's L3 cache which acts like a N-way set-associative cache. Physical RAM is assigned (equally) to these slices using an undisclosed hash algorithm [17–19, 29]. This design allows more cache pages to be reserved by EncExec since cache slices operate mostly independently [29]. Even though these improvements allow EncExec to support a larger working set, the reserved cache is still not enough for complex programs. For these programs, the developers should use EncExec to protect only the sensitive data. Most cryptographic algorithms have a small working set that fits in EncExec's reserved cache.

Large Page Sizes: EncExec controls all the physical pages cached by the reserved cache. For example, our prototype reserves one physical page every 128 pages. This precludes the use of larger pages in the kernel. As previously mentioned, x86 processors support several page sizes, including 4KB, 2MB, and 1GB. They often have separate TLB (translation look-aside buffer) entries for small pages and large pages. Using large pages can thus reduce the TLB pressure for small pages. The kernel uses 2MB pages to map its own code and data. However, EncExec has to reserve 4 small pages from every 2MB page. A kernel with EncExec therefore cannot use large pages. In our prototype, we break large kernel pages into small ones and reclaim the pages we need to reserve. There are two possible workarounds for the kernel to continue using large pages. First, we can compile the kernel so that no code or data will be allocated to the reserved page. The kernel still maps itself with large pages, but none of the reserved pages are actually accessed at runtime. This leaves a number of unused holes in the kernel's address space. As long as these pages are not touched by the kernel, they will not conflict with EncExec. Second, we can restore kernel large pages when EncExec is not in use. The user may not always need the protection of EncExec. For example, he may use EncExec when accessing his bank accounts but not when browsing random Internet sites. This solution will eliminate EncExec's idle performance overhead.

In addition, some I/O devices (e.g., graphic cards) may use large continuous blocks of physical address space for memory-mapped I/O (MMIO). MMIO accesses the device's (on-board) I/O memory instead of the RAM. Memory-mapped I/O will not interfere with EncExec because I/O spaces are often configured to be uncachable in order to correctly interact with I/O devices. Reading/writing I/O memory thus will not cause cache fill or eviction.

Intel SGX: Intel SGX is a powerful and complex extension to Intel CPUs. It creates a trusted execution environment, called enclave, for trusted apps. The enclave's code and data are encrypted in the memory and only decrypted in the CPU cache. SGX's TCB (trusted computing base) consists of only the CPU and the app itself. Therefore, the enclave is protected from cold boot attacks, bus snooping attacks, and malicious high-privileged code (e.g., the hypervisor). SGX has many other useful features, such as remote attestation that can ensure the initial integrity of the trusted app. Compared to SGX, EncExec works on the existing commodity Intel and other CPUs with a similar cache architecture, while SGX is only available in the new Intel CPUs. EncExec is also very lightweight: accessing the protected data in EncExec is instant and does not require time-consuming context switches. A context switch in SGX could be very expensive since it has to flush the TLB and perform various checks [5]. EncExec can also support unmodified programs. Moreover, the design of SGX is vulnerable to cache-based side-channel attacks [8]. By protecting data in the reserved cache, EncExec can provide some (limited) protection against cache side-channel attacks targeting that data, even though the side-channel defense is not the focus of this paper.

6 Related Work

Cold Boot Attacks and Defenses: the first category of related work consists of cold boot attacks and defenses. A cold boot attack exploits the fact that frozen DRAM keeps its contents for a relatively long period of time. It has been demonstrated against both desktop computers [14, 27] and mobile phones [23, 28]. A cold boot attack can be launched by either transplanting the frozen memory to a machine controlled by the attacker or booting a small kernel to dump the memory. Most existing defenses focus on re-purposing hardware storage to protect (small) cryptographic keys [10, 21, 22, 26] or execute cryptographic algorithms [12] on the chip. For example, AESSE [21], TRESOR [22], LoopAmnesia [26], and ARMORED [10] protect an AES key in the SSE registers, debug registers, performance counters, and NEON registers, respectively. These "borrowed" registers naturally can only support compact cryptographic algorithms, but they do not have enough space for algorithms like RSA that have larger memory footprints. Compared to this line of work, EncExec can support all these algorithms.

Copker uses the cache-as-RAM technology [9] to run cryptographic algorithms in the cache. It can also support more complex algorithms such as RSA. However, Copker has very high context switch overheads – it has to force the calling CPU core, as well as any cores that share a cache with it, to enter the no-fill mode of caches. This poses a severe limit on the number of concurrent processes that can use Copker. For example, it can only support *one* process at a time on the Intel Core i7 CPU used in our prototype because the L3 cache is shared by all the cores. Most recent and near-future Intel CPUs all

have a similar cache architecture. EncExec does not have these limitations. For example, it can support multiple concurrent processes and has a close to native performance if used properly. Mimosa uses hardware transactional memory to protect private (RSA) keys from memory disclosure [13]. EncExec also supports large RSA keys and can transparently protect the whole data sections. Both EncExec Mimosa require changes to the OS kernel although EncExec's changes are more invasive. On the other hand, Mimosa requires special hardware support (hardware transactional memory); thus it is not applicable to other architectures.

CaSE combines the cache-as-ram technology and ARM TrustZone to create a system that can protect the data from both cold-boot attacks and the compromised operating system [30]. The flexible cache control of the ARM platform allows CaSE to have lower performance overhead than Copker but similar to EncExec. EncExec instead works on the x86 architecture that lacks find-grained cache control. A recent system called RamCrypt [11] uses moving-window based encryption to protect the process data, similar to our second technique. As mentioned before, this technique alone is potentially susceptible to cold boot attacks because the recently-used unencrypted (sensitive) data can be evicted to the memory and become vulnerable to cold boot attacks.

Other Related Work: EncExec can protect the whole process data from cold boot attacks. Overshadow uses the hypervisor-assisted whole process encryption to protect an application from the untrusted OS kernel [6]. PrivateCore vCage is a virtual machine monitor that implements full-memory encryption for guest VMs by actively managing the whole L3 cache [3]. EncExec focuses on protecting applications. It reserves a small portion of the L3 cache and relies on demand paging to support larger protected data. XnR leverages demand paging to prevent an attacker from reading the randomized code [4]. RamCrypt similarly uses that technology to protect the process data from cold boot attacks [11]. HIveS manipulates the CPU's physical memory layout to hide malware in the I/O memory address space to avoid detection by memory forensic tools [31].

7 Summary

We have presented the design, implementation, and evaluation of EncExec, a practical and effective defense against cold boot attacks. EncExec has two key techniques: spatial cache reservation reserves a small block of the L3 cache, and secure in-cache execution uses demand paging to protect sensitive process data. Under the protection of EncExec, the sensitive data are always encrypted in the memory, and the plaintext data are confined to the reserved cache. Consequently, cold boot attacks can only obtain the encrypted data. The evaluation results demonstrate the effectiveness and practicality of EncExec.

8 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This work was supported in part by the US National Science Foundation (NSF) under Grant 1453020. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

1. FreeBSD. https://www.freebsd.org.
2. SSL Library mbed TLS/PolarSSL. https://tls.mbed.org.
3. Trustworthy Cloud Computing with vCage. https://privatecore.com/vcage/.
4. M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.
5. A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems*, 33(3):8, 2015.
6. X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, 2008.
7. J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, 2004.
8. V. Costan and S. Devadas. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf.
9. Eswaramoorthi Nallusamy. A Framework for Using Processor Cache as RAM (CAR). http://www.coreboot.org/images/6/6c/LBCar.pdf.
10. J. Götzfried and T. Müller. ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices. In *Proceedings of 8th International Conference on Availability, Reliability and Security*, Regensburg, Germany, 2013.
11. J. Götzfried, T. Müller, G. Drescher, S. Nürnberger, and M. Backes. RamCrypt: Kernel-based Address Space Encryption for User-mode Processes. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '16. ACM, 2016.
12. L. Guan, J. Lin, and B. L. Jing. Copker: Computing with Private Keys without RAM. In *Proceedings of the 21th Network and Distributed System Security Symposium*, NDSS '14, 2014.
13. L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 3–19, Washington, DC, USA, 2015. IEEE Computer Society.
14. J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Conference on Security*, San Jose, CA, 2008.
15. J. L. Hennessy and D. A. Patterson. *Computer ARchitecture: a Quantitative Approach*. Morgan Kaufmann, 2012.
16. Intel. *Intel 64 and IA-32 Architectures Software Developerś Manual*, Feb 2014.
17. G. Irazoqui, T. Eisenbarth, and B. Sunar. S $ A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing–and Its Application to AES. In *In Proceedings of the 36th IEEE Symposium on Security and Privacy*, pages 591–604. IEEE, 2015.
18. F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level Cache Side-channel Attacks are Practical. In *In Proceedings of the 36th IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

19. C. Maurice, N. Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, 2015.
20. M. K. McKusick, G. V. Neville-Neil, and R. N. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2014.
21. T. Müller, A. Dewald, and F. C. Freiling. AESSE: A Cold-boot Resistant Implementation of AES. In *Proceedings of the Third European Workshop on System Security*, Paris, France, 2010.
22. T. Müller, F. C. Freiling, and A. Dewald. TRESO: Runs Encryption Securely Outside RAM. In *Proceedings of the 20th USENIX Conference on Security*, San Francisco, CA, 2011.
23. T. Müller and M. Spreitzenbarth. FROST: Forensic Recovery of Scrambled Telephones. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, Banff, AB, Canada, 2013.
24. K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda. PrivExec: Private Execution As an Operating System Service. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, Washington, DC, USA, 2013. IEEE Computer Society.
25. A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 2012.
26. 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*, Orlando, Florida, 2011.
27. Lest We Remember: Cold-boot Attacks on Encryption Keys. `https://citp.princeton.edu/research/memory/`.
28. FROST: Forensic Recovery Of Scrambled Telephones. `http://www1.informatik.uni-erlangen.de/frost`.
29. Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser. Mapping the Intel Last-Level Cache. `https://eprint.iacr.org/2015/905.pdf`.
30. N. Zhang, K. Sun, W. Lou, and Y. T. Hou. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, 2016.
31. N. Zhang, K. Sun, W. Lou, Y. T. Hou, and S. Jajodia. Now you see me: Hide and seek in physical address space. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15. ACM, 2015.