CAUSEC: Cache-based Secure Key Computation with (Mostly) Deprivileged Execution

Shariful Alam*, Le Guan†, Zeyu Chen‡, Haining Wang§, Jidong Xiao¶
* Boise State University, Boise, Idaho, USA
† University of Georgia, Athens, Georgia, USA
‡ University of Delaware, Newark, Delaware, USA
§ Virginia Tech, Arlington, Virginia, USA
¶ Rensselaer Polytechnic Institute, Troy, New York, USA

***Abstract*—As cold boot attacks become a realistic threat to cryptographic systems, several defense solutions have been proposed in the past decade to protect cryptographic systems against such attacks. Interestingly, most of these defense solutions are implemented at the kernel level. Yet running them at the kernel level is risky. Given the complexity of these defense solutions, they inevitably introduce vulnerabilities that could be exploited by attackers and then lead to the compromise of the entire operating system. In this paper, we present CAUSEC which avoids storing crypto keys and other sensitive information in the memory and performs key computation in the cache. CAUSEC protects cryptographic systems against cold boot attacks, but is mostly deprivileged to the user mode. Our experimental results demonstrate that CAUSEC secures key computation and incurs reasonable performance overhead: 11.99% in decryption rate and 7.1% in decryption/signing requests processing when incorporated with the Apache web server.**

***Index Terms*—cold boot attack, cache-based computation**

I. Introduction

Cryptographic systems are critical to the security of today's computer systems and communications. Typically, during runtime, cryptographic systems store their encryption keys in the memory, and rely on the operating systems to protect the keys. Unfortunately, such key computation systems are vulnerable to cold boot attacks [14]. In cold boot attacks, attackers exploit memory remanence effects exhibited by RAM devices to recover encryption keys from memory even after the machine is powered down. Over the past decade, cold boot attacks have been widely studied [11], [24], [27], [30], [32], [34] and proved to be a realistic threat to memory-based key storage. Not surprisingly, on the defense side, different approaches have been proposed to protect cryptographic systems against cold boot attacks as well as more general memory disclosure attacks. Among these defense solutions, Loop-Amnesia [26] and Tresor [22] propose to store encryption keys in privileged CPU registers, whereas Copker [12] and Mimosa [13] propose to store encryption keys in CPU caches. Interestingly, all these four projects have been implemented in the Linux kernel, either as a kernel patch or as a separate kernel module.

However, the implementation and deployment of defense mechanisms as a kernel patch is intrusive to the existing kernel; running the defense as kernel modules is less intrusive, but it still increases the size of the trusted code base (TCB). Copker introduces nearly 8K lines of C code at the kernel level, and Mimosa introduces nearly 10K lines of C code at the kernel level. Such an increase in the code size, inevitably introduces software bugs and exploitable vulnerabilities. Moreover, because these modules run at the kernel level, the compromise of such modules could lead to the compromise of the entire operating system.

In this paper, we present CAUSEC[1], a system which aims to protect encryption keys against cold boot attacks. Unlike the existing solutions, we further aim to dramatically reduce the trusted code base at the kernel level. More specifically, we deprivilege the key computation program to the user mode when possible, and then trap into the kernel mode only when accessing privileged resources is necessary. When the program runs in the user mode, it relies on some privileged instructions to make sure the cryptographic keys in cache are not leaked to RAM. We leverage and modify an existing work namely Dune [5] to expose the needed privileged instructions to the user-mode program without compromising security. We implement CAUSEC on a Linux system. Our experimental results show that CAUSEC can protect the private keys from cold boot attacks while running from user space. CAUSEC incurs reasonable performance overhead: CAUSEC performs 11.99% fewer RSA decryption operations per second comparing with the original MbedTLS library, and when comparing with Apache server utilizing Openssl engine with the original MbedTLS library, CAUSEC can handle 7.1% fewer requests per second.

Overall, this paper makes the following contributions:

- We present CAUSEC, a cache-based defense mechanism to secure key computation in cryptographic systems. Unlike existing work, CAUSEC is a pure user-space application. We implemented a prototype of CAUSEC and validated its efficacy.
- Our evaluation results show that CAUSEC can effectively prevent key information from being leaked to RAM, defeating potential physical attacks to RAM. CAUSEC is efficient enough for production usages. We also report several lessons we learned throughout our implementation and evaluation.

[1]CAUSEC stands for CAche-based User-level SEcure Computation.

The remainder of this paper is structured as follows. We describe the necessary background information and our threat model in Section II. We detail our design and implementation in Sections III and IV, respectively. We present our evaluation results in Section V. We discuss security analysis of our implementation in Section VI. We survey related work in Section VII, and finally we conclude the work in Section VIII.

II. BACKGROUND

A. Threat Model

In this paper, we operate under the presumption that the adversary can acquire physical access to the computing device. With such a privilege, they may be able to shutdown the machine, take the RAM out of the machine and dump the memory out to other storage devices controlled by the adversary. After the above steps, the adversary would be able to recovery information from the memory, and this is the so called "cold boot attack", as presented in [14].

We assume that the operating system is secure and isn't under adversary's control. Adversary does not possess an account on the computing device. Furthermore, the computing device is devoid of any malware or malicious software.

B. Cache Operating Mode

x86 provides different mechanisms for controlling the caching of the data between processor, cache, and memory.

1) Write Back Memory Type: On x86 systems, a CPU cache is in write-back mode if both bit 29 and 30 of the control register `CR0` are cleared [17]. When the write-back cache is enabled on a CPU, any changes in the CPU cache line do not instantly forward to the system memory. Instead, it waits until an explicit or implicit operation is performed. For example, a write-back cache will be synchronized with the system memory if (1) the CPU cache is full; therefore, some cache lines must be evicted; or, (2) cache consistency mechanisms explicitly trigger the synchronization to maintain cache coherency.

While in write-back mode, a read hit accesses the cache and a write hit updates the cache. On the other hand, a read miss replaces the cache and a write miss fills up the cache lines.

C. Cache Management Instructions

On x86, `INVD` and `WBINVD` instructions are used to invalidate the cache lines from all cache levels [17].

- `INVD` is a privileged x86 instruction that operates on all cache levels to invalidate the cache contents. Executing this instruction will invalidate any modified internal cache lines without writing them back to the system memory. In brief, executing `INVD` instruction will result in erasing the local cache contents without any trace.
- `WBINVD` is also a privilege x86 instruction and achieves the same goal as `INVD` by invalidating the internal modified cache lines from all cache levels with one distinction. `WBINVD` instruction writes back the modified cache lines to the system memory before invalidating them.

D. Intel Virtual Machine Extension (VMX) and Dune

Intel VMX introduces two modes of CPU operations: root mode versus non-root mode. In each mode, there are four privilege levels: from ring 0 to ring 3. The intention is to run the hypervisor in the root mode while running the guest virtual machine (VM) in the non-root mode. The VM itself may have an operating system as well as applications. Therefore, the applications in the VM will be running at ring 3 of the non-root mode, whereas the operating system in the VM, which is also known as the guest OS, runs at ring 0 of the non-root mode. The VM runs natively on the CPU, but will cause a VM exit when some privileged resources are accessed. A VM exit is defined as an event of transitioning from the guest OS to the hypervisor. Inside the hypervisor, a function which is defined to handle each VM exit is called a VM-exit handler.

Dune [5] leverages the Intel VMX extension to expose otherwise privileged instructions (i.e., those can only be accessed in ring 0) to user-space applications. It comprises a small kernel module and a user-level library. The kernel module borrows code from Linux KVM and acts like a mini-KVM, which provides a process-level virtualization. The user-level library enables user-level processes to enter into the non-root mode ring 0 (named Dune mode in the original Dune paper), which is achieved by invoking a function named `dune_enter()`. The user-level library communicates with the kernel module via `ioctl` system calls which are sent to a virtual device file under the `/dev` directory (i.e., `/dev/dune`). Processes not in the Dune mode cannot access privileged resources. We leverage and modify Dune to grant user-space applications with access to cache related instructions to confine private data within L1D.

III. DESIGN

A. Design Goals

The major design goal of CAUSEC is to not store the private key in the memory. Instead, we attempt to store the private key in the cache. In other words, while performing the private key computation, we want to ensure that none of the intermediate states or data end up in RAM. Rather, they should be enclosed safely inside a protected environment. Our implementation is based on a key-encryption-key structure where we use one key to securely encrypt another. After successful initialization, the AES master key is derived from a user password. We use this AES master key to encrypt the plaintext RSA private key and keep that encrypted key in the memory.

Our ultimate goal is to provide decryption and signing services using the private key and preserve the information regarding the private key while performing the requested service. Therefore, the following requirements need to be met.

- To ensure the safety of private key computation information in memory, it is necessary to track it. Techniques such as page coloring or page tagging can be employed to monitor memory pages containing sensitive information by assigning them to specific memory locations. This

allows for identifying memory pages holding sensitive data and removing them from the memory address after computation is completed.
- All the necessary variables, including the plaintext private key, should reside inside that aligned memory address space.
- Context switches suspend the execution of the current process, store the corresponding data into the RAM and start executing another process. Therefore, if a context switch happens during the private key computation, all the sensitive information regarding the key will end up in the RAM. Therefore, once the private key computation starts it cannot be interrupted.
- After the computation, the secure environment should be cleaned up thoroughly.

Like Copker [12], we too leverage the L1d cache to create a secure environment. A chunk of bytes are declared as a static variable and then allot memory for it. The size of this static variable should be chosen carefully so that, (1) it can fit inside the L1d cache, (2) large enough to hold all the intermediate states of the private key computing process.

To meet the third requirement, the decryption or signing service needs to run in atomic mode. Hence, before the decryption/signing service starts, interrupts need to be disabled. Without interrupts, the kernel will not be able to schedule any new tasks to the CPU. In a nutshell, disabling interrupts will disable the context switch. Therefore, the decryption service can run interrupt-free.

B. CAUSEC *Architecture*

CAUSEC utilizes a key-encryption-key strategy where a small key, also known as the master key, is used to encrypt larger keys. Using a smaller key gives CAUSEC the ability to store this master key securely into a privileged register, which an adversary can not access from user space. Therefore, the data encrypted with this master key will remain secure.

1) Master Key Generation: CAUSEC employs TRESOR [22] to derive the master key from the user's password and stores it into debug registers. Thus, our master key is also protected by TRESOR. It is worth mentioning that the user's password must be strong enough to withstand brute force attacks.

During OS booting, the AES master key is derived from the user password and put into the debug registers. The key derivation process happens once during early OS initialization and inside the kernel space. After the key is placed into the debug registers, all memory lines that contain sensitive information, such as fragments of the password or key, undergo a thorough erasure process. This involves overwriting all memory traces of the sensitive information.

On a separate offline isolated machine, we use the previously generated AES master key to perform an AES encryption of all plaintext private keys and copy those encrypted keys to the target machine where we intend to deploy CAUSEC. Thus, this target machine does not contain any plaintext private keys but only the AES encrypted private keys.

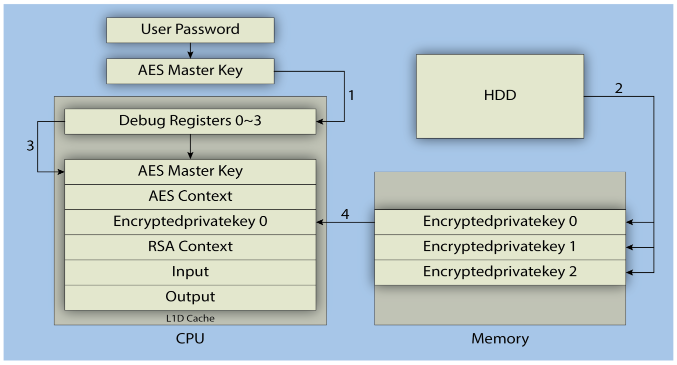

Fig. 1: CAUSEC Key Loading Mechanism

2) Loading Private Key: After the system boots successfully, all encrypted private keys are loaded from the hard drive into RAM. To protect these private keys, they are only decrypted inside the secure environment of the CPU's L1D cache when a request for decryption or signing service is received. At all other times, the private keys remain encrypted in RAM. This ensures that the private keys are only decrypted when necessary, reducing the risk of unauthorized access to the keys. Figure 1 shows a step by step private key loading process.

CAUSEC's private key loading mechanism is similar to Copker [12], (1) During OS boot, TRESOR collects user's password from the command line prompt, generates the master key from this password and copies the master key into the debug registers of all the available CPUs; (2) All the encrypted private keys are loaded from the hard disk drive into RAM; (3) Upon receiving a decryption/signing service request, the master key is loaded from CPU's debug registers into CPU's L1D cache; (4) Corresponding encrypted private key is loaded from the hard disk drive to L1D cache; (5) The AES master key is used to decrypt the encrypted private key, produces a plaintext private key which is then used to perform requested decryption/signing operation.

To adhere to the design goals mentioned in III-A, all these steps, as mentioned earlier, are atomic and can not be interrupted; otherwise, sensitive private key data may end up in the RAM. In addition, these private key-related operations will occur inside the secure environment inside CPU's L1D cache. Output will be written into the RAM only after the private key operations are successfully completed. Prior to releasing the cache, all other variables within the secure environment will be zeroed out. Refer to section III-C for details.

C. Computing within a Secure Environment

In order to ensure that no part of the private key operation will appear in RAM, we leverage the CPU's L1d cache to create a protected environment and perform private key operations in atomic mode inside that enclosed environment. This environment should contain all the variables that the private key operation will use. The minimum essential components that should be in the protected environment are as follows,

- Master key: During the decryption/signing operation, the AES master key will be copied into this master key variable from the debug registers.
- AES context: This AES context variable will contain AES key scheduling information derived from the master key. This information will be used to decrypt the encrypted RSA private key and produce a plaintext RSA private key.
- RSA context: This variable will hold the actual RSA private key. RSA private key is encrypted, and the AES context produces the plaintext RSA private key. RSA context will hold this key.
- Cache stack frame: A stack frame function that operates only within the protected environment will be used by the series functions that perform the RSA private key operation.
- Input/Output: These variables will hold the input and output of the RSA private key operation.

We can not use heap memory inside the protected environment, because heap memory is dynamically allocated, and typically the memory management subsystem in operating systems determines the location of such memory. Therefore it is very difficult to restrict heap memory allocation to a predefined address space. Currently, all standard cryptographic libraries implement RSA using heap memory to handle long integers. This violates our design goal to restrict the RSA private key operation into a secure address space since we can not control the location of the dynamically allocated memory block. We circumvent these hurdles by replacing the heap memory pointer with a static array in the long integer handling function of the cryptographic library. Therefore CAUSEC only uses stack variables during the private key operation.

The operating system controls the stack memory location for each thread. Like heap memory, we can not replace stack memory since all the procedural function call uses the stack. We circumvent this challenge using the same techniques introduced in Copker [12]. During private key operations, CAUSEC takes control of the current stack and creates a new stack inside the CPU's L1d cache, temporarily switches to this newly created stack, and performs the operation. Figure 2 shows the detailed implementation of this customized stack. Copker was built on 32-bit systems, and CAUSEC is built on 64-bit systems, thus our implementation of the stack switch differs slightly from Copker. We will discuss more on this in section IV-B2.

Inside the protected environment, CAUSEC works as follows:

- Read two debug registers and restore the 128-bit AES master key. This key will be used to decrypt the encrypted RSA private key.
- Initialize AES context using the reconstructed AES master key. This context now contains all the AES round keys information. This information will be used to perform an AES decryption.
- Using the AES context, perform an AES decryption on the encrypted RSA private key and produce a plaintext RSA private key.
- The plaintext RSA private key is then loaded into the RSA context to perform the private key operation.
- Perform private key operation on the message using the RSA context and flush the output to the output variable of the secure environment.
- After the private key operation, erase the protected environment before releasing the cache.

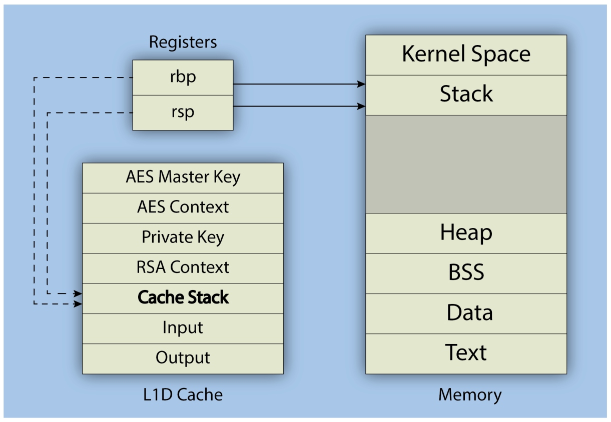

Fig. 2: Customized Stack

1) Atomicity: One of CAUSEC's design goals is, it can not be stopped in the middle of execution. Otherwise, data will end up in RAM which breaks our design goal. In modern operating systems, multi-tasking is common and achieved via context switching. CPU's current state and all its registers contents are stored in RAM during a context switch. Context switches between processes typically happen due to interrupts. Therefore, disabling interrupts will prevent OS from scheduling new tasks to the current CPU.

2) Erase Secure Environment: After responding to a decryption/signing request, it is crucial to clean up secure environment carefully to prevent any sensitive values from ending up in RAM. CAUSEC uses stack memory to create this secure environment, making it easy to erase as the memory is sequential. A loop is run through the predefined stack memory addresses and put zero in each address to overwrite any sensitive data before leaving the atomic section.

IV. Implementation

We developed a CAUSEC prototype in a Linux environment. Our machine runs with Linux kernel 4.4.0 for the x86-64 bit platform. Our OpenSSL [3] version is 1.1.1c. We used Intel Core i7-6700 Quad core CPU with 16GB of RAM for the implementation and validation. CAUSEC utilizes the CPU's L1d (data cache) to create a secure environment. The size of the L1d cache is 32KB.

We implemented CAUSEC as an OpenSSL RSA engine component, and our current implementation contains a total of 8,656 lines of C code. Among these, our RSA engine code contains 1,287 lines of C code, and the rest of the code comes from the cryptography implementation and the Dune user-space library. For the cryptography implementation, we use MbedTLS [2] version 1.2.5. This library was previously known as PolarSSL, and later gets acquired by ARM and renamed MbedTLS.

All the code runs in user space. On the other hand, Copker contains roughly 8k lines of C code that includes the kernel module itself and the MbedTLS library code. Its worth mentioning that, Copker runs entirely inside the kernel except an OpenSSL wrapper.

Currently, we support up to 2048-bit RSA key pairs. The AES master key is generated from the user password during machine booting and stored into the debug registers via TRESOR [22]. We use a 128-bit AES master key to support key encryption key structure. Theoretically, CAUSEC can support up to 256-bit AES master key since each of the four(`db0-db3`) debug register is 64-bit long on an x86-64 bit system. CAUSEC only reads the first two debug registers to reconstruct the 128-bit AES master key. It is important to note that we strictly follow the design goals mentioned in section III-A when implementing CAUSEC.

A. *Exposing Privileged Instructions into User Space*

To better control the cache, CAUSEC needs to access privileged instructions in user space. We adopt Dune [5] and modify it to achieve this goal without compromising security. More specifically, we run user applications in the non-root mode (thanks to the VMX extension) ring 0. Although the user process can now access privileged instructions, a kernel module in the root mode is in place to mediate the access. Concretely, the kernel module configures the virtual machine control structure (VMCS) to enforce which resources are exposed to the user process (in the non-root mode). In our implementation, we allow user processes to access debug registers to retrieve the master key information.

We also added two VM-exit handlers to handle `INVD` and `WBINVD` instructions, which allow user-level processes to invalidate caches. Finally, the default Dune module already allows user-level processes to run `cli/sti` instructions to enable/disable interrupts.

Lessons Learned: We implemented two VM-exit handlers, one to handle `INVD` and the other to handle `WBINVD`, but we later found that the latter is not necessary. This is because the Intel software developer manual [17] classifies these two instructions into two different categories. `INVD` falls into the category of "Instructions That Cause VM Exits Unconditionally", whereas `WBINVD` falls into the category of "Instructions That Cause VM Exits Conditionally". For `WBINVD`, Intel VMX actually allows us to control whether or not running `WBINVD` from the guest OS would trigger a VM exit: the VMCS data structure contains a 32-bit vector called the "secondary processor-based VM-execution controls", and bit 6 of this vector decides whether or not executions of `WBINVD` cause VM exits. In our code, we set this bit to 0, which gives guest OS[2] the control of this instruction, and thus, running this instruction from the guest OS will not trigger a VM exit. For `INVD`, its execution will trigger a VM exit regardless of how the VMCS data structure is configured. Therefore, adding a VM-exit handler into the Dune module is necessary. When handling an exit caused by `INVD`, we first move the guest RIP pointer one instruction forward, and then run `INVD` on behalf of the guest OS.

Dune currently includes 3,973 lines of C code in kernel. To be aligned with our goal of reducing kernel TCB, we manually debloated it by roughly 25%. The removed code mostly includes code that exposes resources CAUSEC does not need, as well as code that is for compatibility purposes. Eventually we introduced approximately 3,000 lines of code into the kernel, which is much less than Copker (8k lines) and Mimosa (10k lines).

1) *Security Consideration of Exposing Privileged Instructions to User Space:*

Exposing privileged instructions into user space does pose some security risks. For example, allowing a user level program to enabling/disabling interrupts gives this program the power to monopolize the CPU which could potentially lead to a denial-of-service (DoS) attack. Such risks can be mitigated with a little effort from the system administrator: All Dune communications are conducted via the device file `/dev/dune`, and therefore the system administrator can set up the permission of this file and only allow either specific users or just the system administrator himself/herself to launch applications which intend to run in the Dune mode. Furthermore, the aforementioned hardware-defined data structure VMCS offers fine-grained control over what a process running in the non-root mode can do. By configuring this data structure, we define what instructions are exposed and what instructions are not exposed. As of now, we only expose the instructions we described above to user space. The original Dune module exposes more, including instructions which load the global descriptor table (`LGDT`), load the local descriptor table (`LLDT`), invalidate the process-context identifier (`INVPCID`), invalidate TLB entries (`INVLPG`), and more. Many of these privileged instructions are not necessary to our key computation system, and thus we do not expose them to user space.

2) *Modification to the User Space Dune Library:*

In Dune, the user space library consists of both C code (6.5k lines) and assembly code (500 lines). We build it as a static library. However, linking it with our OpenSSL engine incurs some unexpected errors in relocation. We later found that this was due to referencing a function with a 32-bit signed number which was not enough in position independent code. We solved this problem by using the `movabsq` instruction to load the 64-bit target address into a temporary register and then indirectly access the target. The modifications we have made are summarized in Table I.

[2]In this paper, we are in the context of process-level virtualization, thus the concept of guest OS is actually CAUSEC, which is just an application.

TABLE I: Modified User Space Dune Library

Original Code	Replacement Code
call `on_dune_exit`	movabsq `$on_dune_exit, %rax` call *%rax
lea `dune_syscall_handler`, %rax	movabsq `$dune_syscall_handler`, %rax
lea `dune_ret_from_user_finish`, %rax	movabsq `$dune_ret_from_user_finish`, %rax
call `dune_trap_handler`	movabsq `$dune_trap_handler, %rax` call *%rax

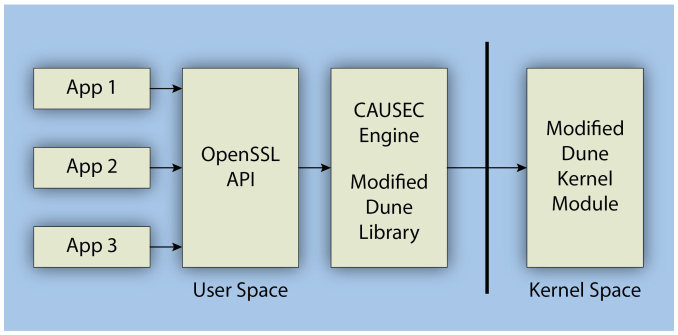

Fig. 3: CAUSEC API Structure

B. OpenSSL RSA Engine

We implemented CAUSEC as an OpenSSL RSA engine component. We show how a CAUSEC-enabled application requests secure cryptographic services and how the requests are routed to the core CAUSEC engine in Figure 3.

Besides, we developed an companion program for CAUSEC to generate and encrypt the private key. It should run offline on a different machine. The encrypted keys should be copied to the target machine (which needs to use the private keys) via out-of-band mechanisms (e.g., USB disk) for security. In the following, we show the process that we use to generate the encrypted private key in the offline machine step by step.

- In the secure offline machine, we use the OpenSSL tool to generate a 2048-bit RSA private key pair.
- We then extract the raw RSA private key component from the RSA private key and store it into a file.
- Our program takes the user's password and uses the same algorithm as TRESOR to generate the AES master key. Then it reads the raw RSA private key components from the file and uses newly generated AES master key to perform an AES encryption and store the encrypted RSA private key into a file.

1) Execution Environment: We define the secure environment in a static data structure which occupies a contiguous memory region. It contains all the variables that CAUSEC needs for private key operations. The data structure is defined as follows.

```
struct SECURE_ENV {
unsigned char masterKey[128/8];
aes_context aes;
rsa_context rsa;
unsigned char encryptedPrivateKey[KEY_BUFFER_SIZE];
unsigned char cachestack[CACHE_STACK_SIZE];
unsigned char in [KEY_LEN];
unsigned char out[KEY_LEN];
}secure_env;
```

`encryptedPrivateKey` holds the encrypted RSA private key. The memory of the variable `cachestack` is where CAUSEC creates its custom stack and performs the private key operation. The size of this static variable is defined by `CACHE_STACK_SIZE` which is 19.50 KB in our prototype. It is large enough to hold all the intermediate variables during the private key operation. Variables `in/out` hold the input buffer and output buffer of the RSA operation respectively. The size of this data structure is 22.58KB, which easily fits in the L1d cache of our target CPU, which is 32KB in size.

2) Custom Stack: To ensure all sensitive data is protected from cold boot attacks, before CAUSEC responds to a decryption/signing request, we create a custom stack frame inside the aforementioned secure environment. More specifically, we write assembly code to manually manipulate both `rsp` and `rbp` to switch the stack to `secure_env.cachestack` defined before. To avoid breaking the code, after each private key operation, we restore both registers to the system allocated ones. This is implemented in the function `stackswitch()` as shown Listing 1.

```
pushq %%rbp
movq %%rsp, %%rbp

//Point to stack bottom.
movq 16(%%rbp), %%rax

// Creating new stack.
movq %%rbp, (%%rax)

// Setting rsp to the new stack
movq %%rbp, -8(%%rax)

// rbx now point to the old rbp
movq %%rbp, %%rbx

// Create new stack frame
movq %%rax, %%rbp
movq %%rax, %%rsp
subq $40, %%rsp

// Parameter for function
movq 32(%%rbx), %%rdx
movq %%rdx, %%rdi

// Call function
call 24(%%rbx)

// Returning to system stack
movq %%rbp, %%rbx
movq (%%rbx), %%rbp
mov -8(%%rbx), %%rsp

leave
```

```
:::rax,rbx
```

Listing 1: stackswitch() function in assembly code

3) RSA Implementation: CAUSEC employs MbedTLS to implement RSA operations. MbedTLS is a portable, modular, lightweight cryptographic library. It implements CRT, sliding window, and Montgomery multiplication to speed up RSA decryption/signing operations. MbedTLS uses heap memory in its long integer module. Since we cannot use heap memory whose location is uncontrollable, we replace the heap memory allocation of a long integer module with a static array. To perform a 2048 bit RSA private key operation, for each long integer, we allocate 268 bytes. To limit the memory consumption, we use a sliding window of size 1.

A cache-line conflict during cryptographic operations could lead to a write-back to memory. To prevent this, the approach strictly limits data usage to the `secure_env`. MbedTLS library ensures thread safety in its cryptographic operations. During RSA operations, all necessary variables, buffers, and states are encapsulated within the `rsa_context` structure. These variables are not shared globally across different RSA operations or instances. In our particular case, we initialize the `rsa_context` within the `secure_env`, effectively isolating and encapsulating all the required variables for cryptographic operations within this secure environment. Thus, cache line conflict will not happen.

4) Loading into L1D cache: To create the secure environment inside the L1d cache, we first need to load the static secure environment mentioned earlier into the L1d cache. We did this by leveraging the write-back memory type.

If an x86 CPU instruction tries to write data into a memory location that has the write-back memory type, the CPU first checks its L1d cache for a cache line containing the memory location of that data. If no such cache line is present in the L1d cache, data is then fetched from the upper-level cache or RAM.

Leveraging this mechanism, we load our secure structure into the L1d cache by reading and writing back one byte at a time from the `secure_env`. The CPU's cache line is filled up with 64 bytes at a time. So, when we read or write one-byte data from an address, the entire 64-byte data is loaded into the cache.

5) Atomicity: To prevent the operating system from relocating the process to another CPU before disabling the interrupt, we set the current process priority to the highest. We did this by using `setpriority()`. Then, we disable hardware interrupts by calling the `cli` instruction which is exposed to the user space with the help of Dune. Therefore, the operating system will not try to schedule any new task to the current CPU. As a result, CAUSEC's execution will run interrupt-free. Upon finishing the assigned task, we call x86's `sti` instruction to re-enable hardware interrupts.

6) CAUSEC *algorithm:* Algorithm 1 shows the primary working mechanism of CAUSEC. Upon receiving a request, CAUSEC uses `sched_getcpu()` to get the core id where the current task is initialized. Then using the processor's affinity, the task is bind to the current core to avoid inconsistency if the task is later scheduled into another core. By calling `dune_enter()`, CAUSEC enters in Dune mode to execute exposed privilege x86-64 instructions. Executing `cli` instruction with inline assembly will disable the interrupt mechanism of the current core. Therefore, OS will not be able to schedule any new task to the current core. Hence, CAUSEC will run interrupt free. Execution of the `sti` instruction will restore the interrupt mechanism.

Algorithm 1: CAUSEC algorithm

```
Global Var: struct SECURE_ENV secure_env, sem_t
            SEM_GLOBAL
Input     : message, encryptedPrivateKey
Output    : to
1  Procedure decrypt() or sign()
2      sem_open (SEM_GLOBAL);
3      $cpuID \leftarrow$ sched_getcpu();
4      Set processor affinity to cpuID;
5      setpriority (PRIOPROCESS, 0, -20);
       // enter dune mode
6      dune_enter ();
7      sem_wait (SEM_GLOBAL);
       // disable interrupts
8      asm ("cli" ::: "memory");
9      fillL1 (secure_env, sizeof(SECURE_ENV));
10     (secure_env $\rightarrow$in) $\leftarrow$ $message$;
11     (secure_env $\rightarrow$encryptedKey)
         $\leftarrow$ $encryptedPrivateKey$;
12     stackSwitch
         (secure_env,secure_operation,secure_env
         $\rightarrow$cacheStack+Cache_Stack_Size$-8$);
13     clearenv (secure_env);
       // enable interrupts
14     asm ("sti" ::: "memory");
15     to $\leftarrow$ (secure_env $\rightarrow$ $out$) ;
16     sem_post (SEM_GLOBAL);
17     sem_close (SEM_GLOBAL);
```

`secure_operation()` performs the secure private key operation upon request using `stackSwitch()`. As mentioned earlier, `stackSwitch()` creates a new stack inside the L1d cache of the current core and the bottom of that stack is pointed by `SECURE_ENV` $\rightarrow$ cacheStack + `CACHE_STACK_SIZE` - 8.

V. Evaluation

This section presents the validation mechanism and performance evaluation results of CAUSEC. Validation provides an experiment-based investigation that confirms that CAUSEC can confine sensitive data in the L1d cache. Performance evaluation provides the experimental results to show the efficiency of CAUSEC when integrated with a real-world application.

A. Security Validation

Proving CAUSEC can hold the data into the L1d cache without flushing it into the RAM is a daunting task due to the lack of instructions to query the cache line along with the absence of cache control utility on x86 platforms [22], [23].

Therefore, to validate CAUSEC, we rely on an experiment-based proof. We adopt a similar validation mechanism that Copker presents. The basic validation idea remains the same except, we did the validation from user space. Unlike Copker [12], CAUSEC executes required privileged x86 instructions for validation purposes from user space by leveraging the small kernel module that we introduce in Section IV.

The validation in its simplest form is described as follows. (1) Take a copy of the current state of the main memory before running any private key operations that use CAUSEC; (2) After the private key operation, invalidate all the modified cache lines before releasing the cache; (3) Finally, compare the current state of the main memory with the previous copy. Any changes in the main memory would indicate a cache leak during the private key operation. Therefore sensitive data has been written into the main memory before invalidating the cache line, thus changing the main memory. On the other hand, an unchanged main memory will verify the validity of CAUSEC.

In our experiment, instead of capturing a copy of the main memory, we inserted specific, fixed-length words into a secure structure. After completing the private key operation, we used the x86 `INVD` privilege instruction to clear any modified cache lines. We then checked the secure structure for the presence of those fixed-length words. If they remained unchanged, it would serve as evidence of the validity of the project.

CAUSEC's validation algorithm is almost the same as the actual working algorithm with a few extra steps. Since the primary goal is to ensure that data inside the L1d cache is never flushed into the RAM while doing the private key operation, we used the `INVD` instruction to invalidate all the modified cache lines before releasing the cache.

However, running the `INVD` instruction calls for a meticulously planned sequence due to its potential to cause data loss by discarding uncommitted data in the cache. Such a situation could create inconsistencies between the cache and RAM, which could, in turn, potentially result in an operating system crash. To avoid such issues, it is imperative to carefully design a sequence that guarantees the successful transfer of all essential modified data back to memory before executing the `INVD` instruction.

We detail the steps of the validation mechanism as follows.

- During the initialization of CAUSEC, we assign a fixed-length word to all the members of the `secure_env` except `in`, `out`, and `encryptedPrivateKey`
- We then execute `WBINVD` instruction to flush all the existing modified cache lines into the memory.
- In order to prevent the memory inconsistency in the current core after executing `INVD` instruction, CAUSEC executes `WBINVD` instruction one more time before calling the `secure_operation()`.
- After the private key operation is completed, we clear the output and use the `INVD` instruction to invalidate any cache lines that were modified during the operation. This ensures that any sensitive information from the private key operation is not stored in cache memory.
- We then check the copy of the `secure_env` in the main memory to see if any of those previously assigned words are modified or not. If those fixed words appear unchanged, we can say that no data has been leaked into the main memory during the private key operation.

In the `secure_env`, the size of the variable `out` is aligned with `CACHE_LINE_SIZE` to prevent flushing data more than the size of the output. We ran the validation algorithm multiple times, and each time we noticed that these fixed words remained unchanged. Thus we are assured that the data will remain locked inside the L1d cache while CAUSEC is running.

The steps outlined earlier serve to demonstrate the validation mechanism. It's important to note that executing both `INVD` and `WBINVD` instructions is a resource-intensive process. We employ these instructions solely to substantiate the authenticity of the experiment. However, in the original algorithm, these instructions are not utilized by CAUSEC.

Lessons Learned: While doing the validation, the execution of the `INVD` instruction was causing the system to stall. Later we found out that hyper-threading was causing this issue. Hyper-threading gives the operating system the illusion that it has more CPUs than its actual number of physical CPUs. Therefore, our system stalls when running the `INVD` instruction on these logical CPUs. We turned off hyper-threading by executing *echo off >/sys/devices/system/cpu/smt/control*.

B. Performance Evaluation

This section reports the performance comparison by examining CAUSEC using the revised MbedTLS library with a plain implementation of OpenSSL engine utilizing the original MbedTLS library running in the same environment for a lateral comparison. The revised MbedTLS library uses a static long integer and a smaller size sliding window (size of 1). On the other hand, everything remains unchanged in the original MbedTLS library implementation. It is worth mentioning that the revised MbedTLS library itself does not promise that sensitive data will prevail in the cache. However, it is one of many pieces that CAUSEC requires to ensure that sensitive data stays in the cache. The Original MbedTLS engine is not exercising any of the steps that CAUSEC is using to enforce security.

As previously stated, CAUSEC is developed as a component of the OpenSSL RSA engine and can be invoked through API calls with OpenSSL. In this experiment, we use a 2048-bit key to evaluate the performance of CAUSEC.

1) Decryption Rate: This study compares the RSA decryption rate of CAUSEC and the original MbedTLS engine. We developed a test program that utilizes OpenSSL's API to invoke both engine implementations, perform 1k decryption, and measure the time required to complete the task. The results of this comparison are presented in

Figure 4, shows the decryption rate comparison of CAUSEC with the Original MbedTLS.

CAUSEC has a lower decryption rate compared to the original MbedTLS. Specifically, CAUSEC can achieve a

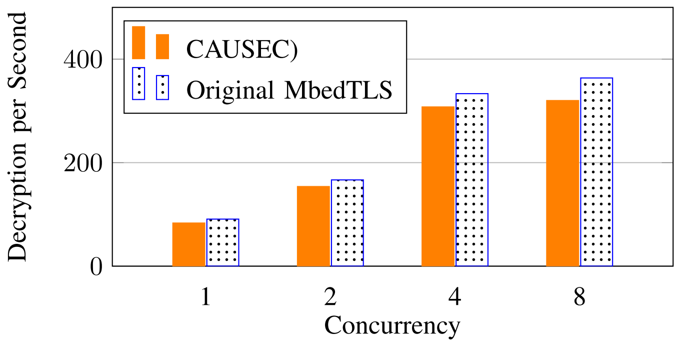

Fig. 4: RSA Decryption Rate

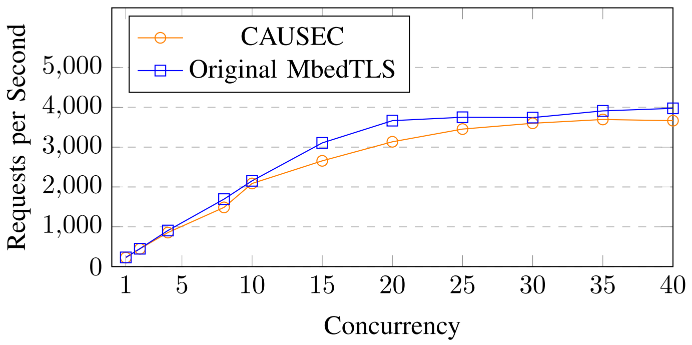

Fig. 5: Apache Benchmark

maximum of 320 RSA decryption operations per second, while the original MbedTLS can achieve 363.63 RSA decryption operations per second. This difference is due to the additional security measures implemented in CAUSEC to protect against cold boot attacks, which result in a reduction of 11.99% (when the concurrency level is 8) in decryption rate as compared to the original MbedTLS.

Lessons Learned: We wrote a test program to test the decryption rate of CAUSEC. This test program is also an OpenSSL engine and only provides decryption. In our original implementation, we were mistakenly using a 32-bit integer in the MbedTLS's bignum.c library and using that library in our 64-bit system. It resulted in a poor decryption rate. We later found the cause and fixed it. We also wrote a shell script that runs the OpenSSL command to invoke our engine via OpenSSL and perform decryption. Inside our engine we were initializing Dune and, after the decryption, forcefully exiting the Dune mode by calling exit(0). Each decryption was taking too much time, since initializing Dune is costly and we were initializing Dune for each decryption.

In our current implementation, after initializing Dune, we start a timer. Then we call a function that implements the Algorithm 1. In this function, we put `stackSwitch()` inside a loop. This loop determines how many decryption operations we want to perform. In our test case, we run this loop 1k times. After all these decryption operations finish, we stop the timer and measure how much time elapsed to determine the decryption rate.

2) Application Level Performance: When combined with a real-life application, we also measure the performance of CAUSEC. We integrate CAUSEC with an Apache Web Server to provide HTTPS services to clients and then measure the throughput of the HTTPS server. We configure our server to serve 1KB of HTML web page via the HTTPS protocol with TLSv1.3 using the `TLS_AES_256_GCM_SHA384` cipher suites. We use ApacheBench [1] to issue 1K requests from a client machine. From the client machine, we generate 1k requests 10 times to the server running both CAUSEC and the original MbedTLS engine. Then we took the average of requests per second for each attempt. Figure 5 shows a comparison of average HTTPS throughput between CAUSEC and an original MbedTLS engine.

CAUSEC has a maximum capability of handling an average of 3695.08 requests per second, while the original MbedTLS can handle 3910.91 requests per second. The gap in performance can be attributed to the implementation of additional security measures, such as the use of the revised MbedTLS library and additional steps to protect against cold boot attacks. These measures lead to an increase in computation time, resulting in a 7.1% decrease in the number of requests handled per second (maximum capability), when compared to the original MbedTLS engine.

Lessons Learned: The Apache server calls CAUSEC to handle decryption/signing requests. In our original implementation, we called `dune_init_and_enter()` to initialize Dune in CAUSEC, but it turns out that a process can not enter into Dune mode multiple times. In the context of Apache, this process is the Apache server process(i.e., httpd), and each of these httpd processes calls CAUSEC. Thus calling `dune_init_and_enter()` in CAUSEC in this situation does not make sense, and our Apache server simply failed. To address this problem, we moved the `dune_init_and_enter()` function call into Apache's source code, inside the server/main.c:main() function, which is the main entry point in the Apache server code. And then in our engine, we call `dune_enter()`, which allows a process to enter into Dune mode. With such a change, the Apache main process enters Dune mode once, and each httpd process also enters Dune mode once.

C. Discussion

CAUSEC uses the CPU's L1d cache to create a secure environment to perform the private key operation to protect against cold boot attacks. The experiment-based validation mechanism demonstrates that sensitive data remains locked inside the L1d caches during the private key computation.

The CPU we used in this work has three levels of cache. While L1 and L2 cache is private to each core, the L3 cache is shared among all the cores. The shared L3 cache architecture in many modern x86 CPUs poses a challenge for us to protect data inside the L1 cache, especially in the presence of increasing memory pressure, resulting in data eviction from the L1 cache to RAM. In order to address this issue, there are two potential approaches:

- Using the `no-fill` mode. The `no-fill` mode is a processor cache control feature that can be enabled on each CPU core through the use of the control register `CR0`. This limits access to the data already presenting in the CPU cache. To prevent sensitive data leaking from the L3 cache, while CAUSEC is running on one CPU core, other CPU cores should be placed into the `no-fill` mode. However, being forced into the `no-fill` mode, applications running on other cores could suffer significant performance degradation due to the lack of L3 cache.
- Leveraging new techniques provided by modern CPUs. A technique called "cache allocation technology" (CAT) [16] was introduced in Intel's Xeon scalable CPUs such as the Haswell processors [16] and has since been available on many Intel processors. CAT prevents data from being evicted from the cache. It uses a bitmask to allocate a specific number of cache ways for each logical processor, which ensures that the data used by a particular application is not evicted by other applications or processes running on the system. In other words, CAT enables cache isolation and can therefore prevent data evictions from shared L3 cache by allowing the user to set a minimum amount of cache that a process is guaranteed. Therefore, implementing CAUSEC in a system with the CAT feature will protect sensitive from being evicted, and at the same time, avoid the performance degradation the first approach introduces. We will leave this as our future work.

VI. Security Analysis

In this section, we provide a theoretical security analysis of attacks on CAUSEC. We do not consider any OS-level attacks on CAUSEC since we assume our OS is secure and free from malware.

A. *Attacks on* CAUSEC

CAUSEC uses the AES master key to decrypt the encrypted RSA private key inside the secure environment of the L1d cache. Therefore, it is crucial to ensure that this master key remains protected against cold boot attacks. To derive this master key, CAUSEC employs TRESOR. Hence, all the security features of TRESOR also apply to CAUSEC. In particular, during early boot, TRESOR generates the master key from the user's password and stores it into debug registers. As such, the master key is safe against cold boot attacks. Finally, the buffer that holds the user's password and all the memories associated with these key derivation processes are erased carefully.

B. *Hardware-Level Attacks*

We also consider an attack on hardware where CAUSEC is deployed and an adversary tries to launch a cold-boot style attack with a malicious booting USB device to extract the cache contents. The idea is if cache lines are not cleared after a system reboot, and the knowledge of the physical address of corresponding cache lines will disclose the contents of the cache. However, this approach does not work. During power-up or reboot, internal cache contents get invalidated. Also, even if the data prevail in the cache due to some hardware features of the cache, a read or write operation would fetch data from the RAM and override the existing cache content.

VII. Related Work

A. *Deprivileged Execution*

It is not hard to understand why there are projects which aim to move lower level code into higher level, as this complies with the least privilege principle - a very basic security principle. In DeHype [33], the authors propose to move a part of the Hypervisor code from kernel level to user level. In the context of QEMU/KVM, this basically means move code from the kernel level component which is KVM, to the user level component, which is QEMU. In SUD [6], the authors propose to move device drivers from user space to kernel space. In MicroDrivers [9], drivers are split into a privileged kernel space code and an unprivileged user space code. RVM [31] introduces a reference validation mechanism which allows device drivers to execute in user space with constrained privileges. The motivation behind this body of research is, device drivers are a significant source of bugs in mainstream operating systems, including Linux and Windows [18]. Some of these bugs allow attackers to take control of the entire operating system. If device drivers are running in user space, then even if they are controlled by a malicious adversary, the rest of the system may still be safe. In CAUSEC, we share a similar motivation with these efforts.

B. *Memory-less Encryption*

AESSE [21], TRESOR [22] and Loop-amnesia [26] propose full disk encryption to protect against cold boot attacks by implementing AES inside the Linux kernel using CPU registers. AESSE uses Streaming SIMD Extensions (SSE) as a key storage [28]. However, it causes many multimedia, math, and 3d applications to break binary compatibility. Furthermore, due to the shortage of space inside CPU registers, the performance of AESSE is six times slower than standard AES implementation. TRESOR uses debug registers to store AES keys and provides AES encryption using x86's `AES-NI` instructions. Loop-amnesia stores secret keys into machine-specific registers (MSRs). Instead of protecting the cryptographic keys from kernel space, CAUSEC leverages TRESOR and protects cryptographic keys from user space using CPU caches. PRIME [10] and RegRSA [35] both use the AES key as a key-encryption key protected by TRESOR and provide RSA implementation in registers. Both of these approaches are implemented inside the kernel. PRIME implements a 2048-bit RSA using Intel's AVX [15] multimedia registers, whereas RegRSA uses vector instructions. Copker [12] and Mimosa [13] are developed as kernel modules and securely implement RSA inside the CPU cache. Mimosa uses Intel's TSX [25] CPU feature to protect private keys against software memory disclosure and cold boot attacks. On the other hand, Copker does not rely on any special CPU features. Instead, it utilizes cache as RAM (CAR) [19] mechanism to protect against cold boot attacks. CAUSEC adopts a similar approach as Copker while running mostly from the user space.

C. Other Approaches against Cold Boot Attacks

BitArmor [20] is a commercial solution for cold boot attacks. While it provides strong protection for keys when the user is not nearby (e.g., when the computer is in hibernation mode), it still stores keys in the RAM when the computer is in use. White-box cryptography [7] hides fixed secret keys into publicly available software binaries, but it introduces much higher overhead, particularly for asymmetric cryptographic algorithms. Moreover, if the binary is stolen, the attacker can still encrypt/decrypt messages with the embedded keys.

Intel Software Guard Extensions (SGX) is a CPU feature that provides a reverse sandbox for high-value applications, even in the presence of a malicious operating system and a compromised BIOS [4], [8]. In its threat model, only the CPU is trusted; therefore, the RAM must be confidentiality- and integrity-protected. To this end, it adopts a hardware-enforced cryptographic mechanism to transparently encrypt all the data traffic from the CPU to the RAM, and decrypt it when the data flows back to the CPU. As a result, SGX-protected applications are immune to cold boot attacks by nature. However, it is subject to side-channel attacks [29], and notable performance overhead has been observed [4].

Moreover, it only supports newer Intel CPUs while CAUSEC generally applies to Intel's Pentium 4 and later and AMD's Athlon 64 and later processors. In particular, any processor with cache and debug registers.

VIII. Conclusion

We presented CAUSEC, a key computation system which provides decryption and signing services for other applications. CAUSEC stores sensitive key information in the L1d cache instead of the main memory. CAUSEC protects cryptographic systems against cold-boot attacks, but unlike several state-of-the-art defense systems which perform key computation mainly in the kernel level, CAUSEC has a much smaller attack surface in the kernel space. CAUSEC accomplishes this goal via leveraging a small modified Dune kernel module, which establishes a process-level virtualization environment and exposes certain privileged instructions into user space. As such, CAUSEC is able to access privileged resources such as debug registers, interrupt flags, and cache invalidation instructions. Our experimental results demonstrate that CAUSEC provides the promised security and incurs reasonable performance overhead - 11.99% in decryption rate and 7.1% in decryption/signing requests processing when incorporated with the Apache web server.

IX. Acknowledgements

We would like to thank the anonymous reviewers for their insightful feedback, which helped us improve the quality of this paper. This work was supported in part by the U.S. Army Research Office (ARO) grant W911NF-19-1-0049 and Office of Navy Research (ONR) grant N00014-23-1-2158. Opinions and conclusions presented in the paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] ab - Apache HTTP Server Benchmarking Tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2023. [Online; accessed 20-January-2023].

[2] Mbed TLS. https://tls.mbed.org/, 2023. [Online; accessed 20-January-2023].

[3] OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/, 2023. [Online; accessed 20-January-2023].

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 16, pages 689–703, 2016.

[5] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazieres, and Christos Kozyrakis. Dune: Safe User-Level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–348, 2012.

[6] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *USENIX annual technical conference*, volume 2010. Boston, 2010.

[7] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-box Cryptography and an AES Implementation. In *International Workshop on Selected Areas in Cryptography*, pages 250–270. Springer, 2002.

[8] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[9] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The Design and Implementation of Microdrivers. *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 43(3):168–178, 2008.

[10] Behrad Garmany and Tilo Müller. PRIME: Private RSA Infrastructure for Memory-Less Encryption. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, pages 149–158, 2013.

[11] Michael Gruhn and Tilo Müller. On the Practicability of Cold Boot Attacks. In *Proceedings of the 8th International Conference on Availability, Reliability and Security (ARES)*, pages 390–397. IEEE, 2013.

[12] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. Copker: Computing with Private Keys without RAM. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*, pages 23–26, 2014.

[13] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting Private Keys against Memory Disclosure Attacks Using Hardware Transactional Memory. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, pages 3–19. IEEE, 2015.

[14] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest We Remember: Cold-Boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium*, pages 45–60. USENIX, 2008.

[15] Intel. Advanced Vector Extensions Programming Reference. *Intel Corporation*, 2011.

[16] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *Intel Corporation, April*, 2015.

[17] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A*, December 2016.

[18] Asim Kadav and Michael M Swift. Understanding Modern Device Drivers. *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 47(4):87–98, 2012.

[19] Yinghai Lu, Li-Ta Lo, Gregory R Watson, and Ronald G Minnich. CAR: Using Cache as RAM in Linux BIOS.

[20] Patrick McGregor, Tim Hollebeek, Alex Volynkin, and Matthew White. Braving the Cold: New Methods for Preventing Cold Boot Attacks on Encryption Keys. In *Black Hat Security Conference*, 2008.

[21] Tilo Müller, Andreas Dewald, and Felix C Freiling. AESSE: A Cold-Boot Resistant Implementation of AES. In *Proceedings of the Third European Workshop on System Security*, pages 42–47, 2010.

[22] Tilo Müller, Felix C Freiling, and Andreas Dewald. TRESOR Runs Encryption Securely Outside RAM. In *Proceedings of the 20th USENIX Security Symposium*, volume 17, 2011.

[23] Jürgen Pabel. Frozencache: Mitigating Cold-Boot Attacks for Full-Disk-Encryption Software. In *27th Chaos Communication Congress*, 2010.

[24] Bertram Poettering and Dale L Sibborn. Cold Boot Attacks in the Discrete Logarithm Setting. In *Cryptographers' Track at the RSA Conference*, pages 449–465. Springer, 2015.

[25] Ravi Rajwar and Martin Dixon. Intel Transactional Synchronization Extensions. In *Intel Developer Forum San Francisco*, volume 2012, 2012.

[26] Patrick 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 (ACSAC)*, pages 73–82, 2011.

[27] Benjamin Taubmann, Manuel Huber, Sascha Wessel, Lukas Heim, Hans Peter Reiser, and Georg Sigl. A Lightweight Framework for Cold Boot Based Forensics on Mobile Devices. In *Proceedings of the 10th International Conference on Availability, Reliability and Security (ARES)*, pages 120–128. IEEE, 2015.

[28] S Thakkur and Thomas Huff. Internet Streaming SIMD Extensions. *Computer*, 32(12):26–34, 1999.

[29] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 339–354. IEEE, 2021.

[30] Tian Wang, Xiaoxin Cui, Yewen Ni, Dunshan Yu, Xiaole Cui, and Gang Qu. A Practical Cold Boot Attack on RSA Private Keys. In *Proceedings of the 2017 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 55–60. IEEE, 2017.

[31] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B Schneider. Device Driver Safety Through a Reference Validation Mechanism. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 8, pages 241–254, 2008.

[32] Yoo-Seung Won, Jong-Yeon Park, Dong-Guk Han, and Shivam Bhasin. Practical Cold Boot Attack on IoT Device - Case Study on Raspberry Pi. In *Proceedings of the 2020 IEEE International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*, pages 1–4. IEEE, 2020.

[33] Chiachih Wu, Zhi Wang, and Xuxian Jiang. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS)*, pages 1–15. Citeseer, 2013.

[34] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd Austin. Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 313–324. IEEE, 2017.

[35] Yuan Zhao, Jingqiang Lin, Wuqiong Pan, Cong Xue, Fangyu Zheng, and Ziqiang Ma. Regrsa: Using Registers as Buffers to Resist Memory Disclosure Attacks. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 293–307. Springer, 2016.