2015 IEEE Symposium on Security and Privacy

Protecting Private Keys against Memory Disclosure Attacks using Hardware Transactional Memory

Le Guan*†§, Jingqiang Lin*†, Bo Luo‡, Jiwu Jing*† and Jing Wang*†
*Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, China
†State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, China
‡Department of Electrical Engineering and Computer Science, the University of Kansas, USA
§University of Chinese Academy of Sciences, China
Email: {guanle, linjingqiang}@iie.ac.cn, bluo@ku.edu, {jingjiwu, wangjing}@iie.ac.cn

***Abstract*—Cryptography plays an important role in computer and communication security. In practical implementations of cryptosystems, the cryptographic keys are usually loaded into the memory as plaintext, and then used in the cryptographic algorithms. Therefore, the private keys are subject to *memory disclosure attacks* that read unauthorized data from RAM. Such attacks could be performed through *software* methods (e.g., OpenSSL Heartbleed) even when the integrity of the victim system's executable binaries is maintained. They could also be performed through *physical* methods (e.g., cold-boot attacks on RAM chips) even when the system is free of software vulnerabilities. In this paper, we propose *Mimosa* that protects RSA private keys against the above software-based and physical memory attacks. When the Mimosa service is in idle, private keys are encrypted and reside in memory as ciphertext. During the cryptographic computing, Mimosa uses hardware transactional memory (HTM) to ensure that (*a*) whenever a malicious process other than Mimosa attempts to read the plaintext private key, the transaction aborts and all sensitive data are automatically cleared with hardware mechanisms, due to the strong atomicity guarantee of HTM; and (*b*) all sensitive data, including private keys and intermediate states, appear as plaintext only within CPU-bound caches, and are never loaded to RAM chips.**

To the best of our knowledge, Mimosa is the first solution to use transactional memory to protect sensitive data against memory disclosure attacks. We have implemented Mimosa on a commodity machine with Intel Core i7 Haswell CPUs. Through extensive experiments, we show that Mimosa effectively protects cryptographic keys against various attacks that attempt to read sensitive data from memory, and it only introduces a small performance overhead.

I. Introduction

Cryptosystems play an important role in computer and communication security, and the cryptographic keys shall be protected with the highest level of security in computer systems. However, in signing or decryption operations, the private keys are usually loaded into memory as plaintext, and thus become vulnerable to *memory disclosure attacks* that read sensitive data in memory. Firstly, such attacks could be launched through software exploitations. For instance, the OpenSSL Heartbleed vulnerability allows remote attackers without any privileges to steal sensitive data in memory [58]. Malicious unprivileged processes can exploit other different vulnerabilities [31, 50, 57, 59] to obtain unauthorized memory data. According to the statistics of Linux vulnerabilities [24], 16.2% of the vulnerabilities can be exploited to read unauthorized data from the memory space of operating system (OS) kernel or user processes. These memory disclosure attacks can be launched successfully, even when the integrity of the victim system's executable binaries is maintained at all times. Hence, existing solutions such as buffer-overflow guards [22, 23, 83] and kernel integrity protections [38, 47, 63, 70], are ineffective against these "silent" memory attacks. Finally, attackers with physical access to the computer are capable of bypassing all the OS protections to directly read data from RAM chips, even when the system is free of the vulnerabilities mentioned above. For example, the cold-boot attacks [32] "freeze" the RAM chips of a running victim computer, place them into another machine controlled by the attacker, and then read the RAM contents.

In this paper, we present Mimosa, which uses *hardware transactional memory* (HTM) to protect private keys against software and physical memory disclosure attacks described above. In particular, we use Intel Transactional Synchronization eXtensions (TSX) [40], a commodity implementation of HTM in commercial-off-the-shelf (COTS) platforms. Transactional memory is originally proposed as a speculative memory access mechanism to boost the performance of multi-threaded applications [37]. However, we find that the strong *atomicity* guarantee provided by HTM can be utilized to defeat illegal concurrent accesses to the memory space that contains sensitive data. Moreover, TSX and most HTM are physically implemented on top of CPU caches, so that cryptographic computing using TSX can be constrained entirely in the CPU, effectively preventing cold-boot attacks on RAM chips.

In Mimosa, each private-key computation is performed as an atomic transaction. During the transaction, the encrypted private key is first decrypted into plaintext, and then used to decrypt or sign messages. If the transaction is interrupted due

Jingqiang Lin is the corresponding author.

© 2015, Le Guan. Under license to IEEE.
DOI 10.1109/SP.2015.8

to any reason (e.g., attack attempt, interrupt, exception, or fault), a *hardware-enabled* abort handler clears all updated but uncommitted data in the transaction, which guarantees that the sensitive private key (and all intermediate states) cannot be accessed by concurrent malicious processes. Note that the abort processing is *non-maskable*, and is triggered by HTM automatically. Before committing the computation result, all sensitive data are carefully cleared. Hence, in Mimosa, a software-based memory disclosure attack only obtains cleared data, even if it successfully reads from the memory addresses for the cryptographic keys or other sensitive data.[1] Meanwhile, with the Intel TSX implementation, the transaction is performed entirely within CPU caches and the updated but uncommitted contents (i.e., the plaintext private keys) are never loaded to the RAM chips. Therefore, Mimosa is also immune to cold-boot attacks.

When the private keys are at-rest (i.e., there is no signing or decryption request), they always remain encrypted by an AES key-encryption key. Mimosa integrates TRESOR [56], a register-based AES cryptographic engine, to protect the AES master key in debug registers that are only accessible with ring 0 privileges. If Mimosa is triggered for a signing/decryption task, the private key is decrypted by the AES master key, and then used for signing/decryption; the whole process is implemented as a transaction as introduced above.

We have implemented the prototype system with Intel TSX, but the Mimosa design is applicable to other existing HTM implementations using on-chip caches [45, 82] or store buffers [2, 27, 33]. When the private-key computation is executed as a transaction protected by HTM and the private key is decrypted (i.e., the data are updated) on-the-fly in the transactional execution, any attack attempt to access the private key would result in data conflicts that would abort the transaction. Because these HTM solutions are CPU-bound, they are also effective in preventing cold-boot attacks.

Performing the computationally expensive private-key operation as a transaction with Intel TSX is much more challenging than it seems to be. Because transaction memory is originally proposed for speculatively running critical sections, a transaction with Intel TSX is typically lightweight, such as setting or unsetting a shared flag variable. To support RSA private-key operations, the Mimosa computing task needs to address many problems, including unfriendly instructions, data sharing intrinsic in OS functions, local interrupts, kernel preemption, and other unexpected aborts; otherwise, the transactional execution would never commit.

Mimosa is implemented as a kernel module in Linux and exported as an OpenSSL cryptographic engine. We have successfully evaluated the Mimosa prototype on an Intel Core i7 4770S Haswell CPU with TSX. Experimental results show that Mimosa only introduces a small overhead to provide the security guarantees. Its performance is very close to popular RSA implementations without additional security protections. Through extensive validations, we confirm that no private key is disclosed under various memory disclosure attacks.

Our contributions are three-fold. (1) We are the first in the literature to utilize transactional memory to ensure the confidentiality of sensitive information, especially private keys, against software and physical memory disclosure attacks. (2) We have implemented the Mimosa prototype on a commodity implementation of HTM (i.e., Intel TSX), and the experimental evaluation showed that it is immune to the memory disclosure attacks with a small overhead. And (3) we develop an empirical guideline to perform heavy computations in an Intel TSX transaction, which suggests the possibility to extend the applications of HTM.

The rest of the paper is organized as follows. The background and preliminaries are introduced in Section II. We then present the Mimosa's design and implementation details in Sections III and IV, respectively. Experimental results are shown in Section V, and the security analysis is in Section VI. We summarize related works in Section VII and finally conclude the paper.

II. Background and Preliminaries

This section first summarizes the software and physical attacks that steal sensitive data in memory. We then discuss the CPU-bound solutions against cold-boot attacks. Finally, we introduce transactional memory and one of its hardware implementations, Intel TSX, which is used in Mimosa.

A. Memory Disclosure Attacks on Sensitive Data

Memory disclosure attacks are roughly classified into two categories: software-based and hardware (or physical) attacks. Software attacks usually exploit system vulnerabilities to read unauthorized addresses in the memory space, while hardware-based attacks require physical access to the victim machine to read from RAM chips.

Software Memory Disclosure Attack. Various software vulnerabilities allows adversaries to read unauthorized data from the memory space of OS kernel or user processes without modifying kernel binaries. That is, even when the integrity of the victim system is ensured, memory disclosure attacks can be launched successfully.

These memory vulnerabilities result from unverified inputs, isolation defects, memory dump, memory reuse or cross-use, and uncleared buffers. For example, the OpenSSL Heartbleed vulnerability [58] allows remote attackers to receive sensitive data by manipulating abnormal SSL heartbeat requests; or attackers can exploit the vulnerability reported in [31] to read memory at a random location. The uninitialization error [57] and the ALSA bug [59] leads to

[1] Our solution reactively clears the memory to protect sensitive data whenever an attack attempt is detected. Hence, we name it *Mimosa*, as it is similar to the plant *Mimosa pudica*, which protects itself by folding its leaves when touched or shaken.

sensitive information leakage from kernel memory. As a result of unintended software design and implementation issues, such as swap, core dump, hibernation and crash reports, memory content could be swapped to disks [17], which may be accessible to attackers. For example, cryptographic keys are recovered from Linux memory dump files [65]. Some FTP and Email servers dump core files to a directory accessible to adversaries [48, 74, 79, 80], leaking the passwords that were kept in memory. Finally, uncleared data buffers in memory are subject to reuse or cross-use [77, 78]. By exploiting the Linux ext2 implementation vulnerability, private keys of OpenSSH and HTTPS can be exposed from uncleared buffers [35].

Cold-Boot Attack. This typical and powerful physical memory attack results from the remanence effect of semiconductor devices; that is, the contents of dynamic RAM (DRAM) chips gradually fade away. At low temperatures the fading speed slows down significantly. Hence, adversaries can retrieve the remained data by cold-booting the *running* target computer and loading a malicious OS [32]. The cold-boot attacks are launched by resetting the computer and loading a malicious OS from an external storage, or alternatively by placing the DRAM chips into another machine controlled by the attacker. The cold-boot attack requires no account or credential information on the target machine, and can be launched even if the victim system is free of the vulnerabilities that can otherwise be exploited by software memory disclosure attacks.

B. *CPU-Bound Solutions against Cold-Boot Attacks*

While there are different solutions against software memory disclosure attacks [14, 28, 35, 61], the countermeasure against cold-boot attacks is to bound the operations in CPUs. The idea of CPU-bound solutions is to avoid loading sensitive data (e.g., AES keys) into RAM chips, so that cold-boot attacks would fail. Register-based cryptographic engines [55, 56, 73] have implemented the AES algorithm entirely within the processor. In particular, TRESOR [56] stores the AES keys in debug registers and Amnesia [73] uses model-specific registers. These register-based engines also prevent software memory disclosure attacks, because the keys and sensitive intermediate states never appear in memory. Note that atomicity must be ensured in a block encryption/decryption to avoid swapping register states to memory, including general purpose registers that store intermediate values.

PRIME [29] and Copker [30] extends the CPU-bound solutions to asymmetric algorithms. The AES key protected by TRESOR is used as a master key (i.e., the key-encryption key) to encrypt RSA private keys. In PRIME [29], the private key is first decrypted into AVX registers and the RSA computations are performed within these registers. The performance is only about 10% of the traditional implementations, due to the limited size of registers. Meanwhile, Copker [30] employs CPU caches to perform RSA decryption/signing, which results in better performance. However, Copker depends on a trustworthy OS kernel to avoid illegal memory read operations to keep the sensitive keys in caches. That is, Copker is not immune to software memory disclosure attacks.

C. *Transactional Memory and Intel TSX*

Transactional memory is a memory access mechanism of CPUs, originally designed to improve the performance of concurrent threads and reduce programming efforts [37]. Programmers can accomplish fine-grained locking with coarse-grained locks. The key idea is to run critical sections speculatively and serialize them only in the case of data conflicts, which happen when several threads concurrently access the same memory location and at least one of them attempts to update the content. If the entire transaction is executed without any conflict, all modified data are committed atomically and made visible to other threads; otherwise, all updates are discarded and the thread is rolled back to the automatically-saved checkpoint. Transactional memory can be implemented in software [16, 34] or supported by hardware [40, 45, 82].

Intel TSX [40], first shipped in the 4th-generation Core CPUs (i.e., Haswell), provides transactional memory support that is completely *hardware-enabled*. Programmers only need to specify critical sections for transactional execution, the processor transparently performs conflict detection, commit and roll-back operations. To detect data conflicts, Intel TSX keeps all updated but uncommitted data in the first-level data (L1D) cache, and tracks a read-set (addresses that have been read from) and a write-set (addresses that have been written to) in the transaction.

Data conflicts are detected on top of the cache-coherence protocol, at the granularity of cache lines. A data conflict is detected if another core either (*a*) reads from a memory location that is in the transaction's write-set, or (*b*) writes to a location in the write-set or read-set. If no conflict is detected, all write operations within the transaction are committed and become visible to other cores atomically. Otherwise, all updated data are discarded and the thread is rolled back to the saved checkpoint, as if the transaction never started.

However, except for data conflicts, several other events can cause an Intel TSX transaction to abort. This includes unfriendly instructions such as cache-control instructions (e.g., `CLFLUSH` and `WBINVD`), operations on the X87 and MMX architecture states, background system activities such as interrupt and exception, and executing self-modifying codes. There are also other micro-architectural implementation dependent reasons. For a detailed list of events that may abort a transaction, see the Intel TSX specification [40].

Intel TSX provides two programming interfaces with different abort handling mechanisms. First, Hardware Lock

Elision (HLE) is compatible with legacy instructions, and works with two new instruction prefixes (i.e., XACQUIRE and XRELEASE). The prefixes give hints to processors that execution is about to enter or exit the critical section. On aborts, after rolling back to the original state, the processor automatically restarts the execution in a legacy manner; that is, locks are acquired before entering the critical section.

The second TSX programming interface called Restricted Transactional Memory (RTM), provides three new instructions (i.e., XBEGIN, XEND and XABORT) to start, commit, and abort transactional execution. In RTM, programmers specify a *fallback function* as the operand of XBEGIN. Aborted execution jumps to the specified address of the fallback function, so the programmers can implement customized codes to handle the situation; for instance, to retry or explicitly acquire a lock.

III. System Design

In this section, we first present the assumptions and security goals of Mimosa. We then introduce the general system architecture, and some important details in the design of Mimosa.

A. Assumptions and Security Goals

Assumptions. We assume the correct hardware implementation of HTM (i.e., Intel TSX in our prototype system or others in the future). This assumption is expected to be guaranteed in COTS platforms. We also assume a secure initialization phase during the OS boot process; that is, the system is clean and not attacked during this small time window.

The attackers are assumed to be able to launch various memory disclosure attacks on the protected system. The attackers can stealthily read data in memory with root privileges by exploiting software vulnerabilities [31, 50, 57–59], or launch cold-boot attacks [32] on the system. They can also eavesdrop the communication between the CPU and RAM chips on the bus. Mimosa is designed to defend against the "silent" memory disclosure attacks that read sensitive data from memory without breaking the integrity of the systems' executable binaries. For instance, the attacks that exploit various Linux kernel vulnerabilities [24] to access unauthorized data. We do not consider the multi-step attacks that compromise OS kernel – the attacks that first write malicious binary codes into the victim machine's kernel, and then access sensitive data via the injected codes. That is, Mimosa assumes that the integrity of OS kernel is not compromised.

Different from the existing security mechanisms which attempt to detect or prevent software attacks (e.g., kernel integrity protections [38, 47, 63, 70] and buffer-overflow guards [22, 23, 83]), Mimosa follows a different philosophy – it tries to "dance" with attacks. That is, even when an attacker exploits memory disclosure vulnerabilities (e.g., OpenSSL Heartbleed [58]) to successfully circumvent these protections and read data from memory, Mimosa ensures that the attacker still cannot obtain the private keys that were originally stored at the memory address.

Last, since Mimosa employs TRESOR [56] to protect the AES master key, it also inherits the assumptions made by TRESOR. In particular, TRESOR (and similar solutions [29, 30, 73]) assumes an OS without any interface or vulnerability that allows attackers to access the privileged debug registers. As analyzed in [29, 30, 56, 73], the access to the privileged debug registers can be blocked by patching the ptrace system call (the only interface to debug registers from user space applications), disabling loadable kernel modules (LKMs) and kmem, and removing JTAG ports (as done in COTS products).

Security Goal. Based on the above assumptions, we design Mimosa with the following goals:

1) During each signing/decryption computation, no process other than the Mimosa computing task can access the sensitive data in memory, including the AES master key, the plaintext RSA private key and intermediate states.
2) Either successfully completed or accidentally interrupted, each Mimosa computing task is ensured to immediately clear all sensitive data, so it cannot be suspended to dump these sensitive data.
3) The sensitive data never appear on the RAM chips.

The first goal thwarts direct software-based memory disclosure attacks, and the second prevents the sensitive data from being propagated to other vulnerable places. The third goal makes a successful cold-boot attack only get encrypted copies of private keys.

B. The Mimosa Architecture

Mimosa adopts the common key-encryption-key structure. The AES master key is generated early during the OS boot process and is stored in debug registers since then. The RSA context is dynamically constructed, used and finally destroyed within a transactional execution, when Mimosa serves signing/decryption requests. When the Mimosa service is in idle, the private keys always remain encrypted by the AES key.

The operation of Mimosa consists of two phases as shown in Figure 1: an *initialization* phase and a *protected computing* phase. The initialization phase is executed only once when the system boots. It initializes the AES master key and sets up necessary resources. The protected computing phase is executed on each RSA private-key computation request. This phase performs the requested RSA computations. All memory accesses during the protected computing phase are tracked and examined to achieve the security goals of Mimosa.

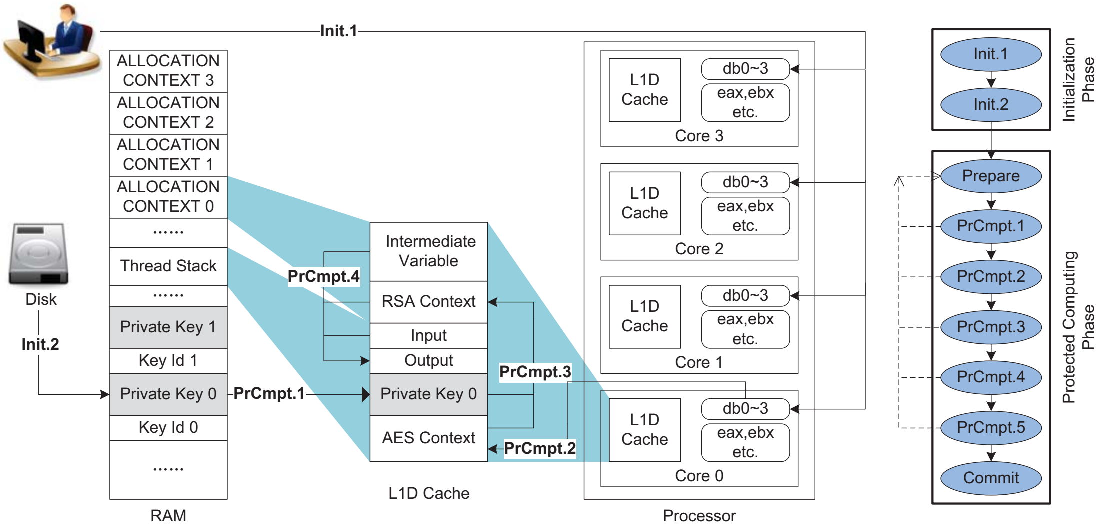

Figure 1: Mimosa Overview

Initialization Phase. This phase contains two steps. **Init.1** resembles TRESOR [56] and executes completely in kernel space when the system boots. First, a command line prompt is set up for the user to enter a password. Then, the AES master key is derived from the password, and copied to the debug registers of every CPU core. All intermediate states of this derivation are carefully erased. Moreover, the user is required to type in 4096 more characters to overwrite input buffers. We assume that there is no software or physical memory disclosure attack during this step, and the password is strong enough to resist brute-force attacks.

In **Init.2**, a file containing an array of ciphertext private keys is loaded from hard disks or other non-volatile storages into memory. These private keys are securely generated and encrypted by the AES master key into the file in a secure environment, e.g., another off-line trustworthy machine.

Protected Computing Phase. When Mimosa receives a private-key computation request from users, it uses the corresponding private key to perform the computation, and then returns the result to users. In this phase, Mimosa prepares the transactional execution, performs the private-key computation, erases all sensitive data, and finally terminates the transaction to commit the result. In particular, it includes the following steps:

- **Prepare**: HTM starts to track memory accesses in the read-set and the write-set in the L1D cache.
- **PrCmpt.1**: The ciphertext private key is loaded from the RAM to the cache.
- **PrCmpt.2**: The master key is loaded from the debug registers to the cache.
- **PrCmpt.3**: With the master key and ciphertext private key, the private key context is constructed.
- **PrCmpt.4**: With the plaintext private key, the requested decryption/signing operation is performed.
- **PrCmpt.5**: All the sensitive variables in caches and registers are erased, except the result.
- **Commit**: Finish the transaction and make the result available.

All memory accesses during the protected computing phase are strictly monitored by hardware. In particular, we declare a *transactional region*. During the transactional execution, all memory operations that might break Mimosa's security principles are detected by hardware: (1) any attempt to access the modified memory locations, i.e., the plaintext private key and any intermediate states generated in the transactional execution; and (2) cache eviction or replacement that synchronizes data in caches to the RAM.

If no such memory exception is detected, the transaction commits and the result is returned to users. Otherwise, the hardware-enabled *abort processing handler* is triggered automatically to discard all modified data. Then, it executes the program-specified *fallback function* (not shown in Figure 1) to process the exceptional situation; in the fallback function, we can choose to retry immediately or take other supplementary actions before retrying (see Section IV-C).

To take full advantage of multi-core processors, Mimosa is designed to support multiple private-key computation tasks in parallel. Each core is configured with its own resources for Mimosa. A block of memory space is reserved for each core in the transactional region (i.e, the protected computing phase). This space is mainly used for the dynamic memory allocation in RSA computations. The reserved space is separated properly for each core to avoid data conflict that would lead to aborts (see Section IV-C for details).

Finally, we would like to emphasize that the design of the Mimosa architecture is based on the general properties of hardware transactional memory. That is, Mimosa does not rely on any specific HTM implementation. It is expected that this architecture could be adopted with any COTS HTM product. In the rest of the paper, we will describe the implementation and evaluation of Mimosa with a COTS HTM product, i.e., Intel TSX.

IV. Implementation

Mimosa is implemented as a kernel module patched to Linux kernel v3.13.1. It provides RSA private-key computation services to the user space through the `ioctl` system call. In addition, the `ioctl` interface is further encapsulated into an OpenSSL engine, facilitating the integration of Mimosa into other applications. The Mimosa prototype supports 1024/2048/3072/4096-bit RSA algorithms.

In this section, we firstly describe the RTM interface of Intel TSX, and then a naïve implementation of Mimosa. However, in this implementation, the transactional execution rarely commits, resulting in unacceptable performance. Next, we identify the abort reasons, and eliminate them one by one. The final implementation offers efficiency comparable to conventional RSA implementations without such protections. The performance tuning steps produce an empirical guideline to perform heavy cryptographic computations with Intel TSX. We also briefly describe the utility issues of Mimosa, including private key generation and the user-space API. Finally, we discuss the applicability of Mimosa design, i.e., how to apply the Mimosa architecture to other HTM solutions.

A. RTM Programming Interface

In the protected computing phase, the computation is constrained in a transaction. Mimosa utilizes Intel TSX as the underlying transactional memory primitive. In particular, we choose RTM as the HTM programming interface. With this flexible interface, we have control over the fallback path, in which Mimosa can define the policy to handle aborts.

RTM consists of three new instructions (`XBEGIN`, `XEND` and `XABORT`) to start, commit and abort a transactional execution. `XBEGIN` consists of a two-byte opcode `0xC7 0xF8` and an immediate operand. The operand is a relative offset to the `EIP` register, which is used to calculate the address of the *program-specified* fallback function. On aborts, TSX immediately breaks the transaction and restores architectural states by the *hardware-enabled* abort handler. Then, the execution resumes at the fallback function. At the same time, the reason of abort is marked in the corresponding bit(s) of the `EAX` register. The reason code in `EAX` is used for quick decisions (in the fallback function) at runtime; for example, the third bit indicates a data conflict, and the fourth bit indicates that the cache is full. However, this returned code does not precisely reflect every event that leads to the abort [40]. For instance, aborts due to unfriendly instruction or interrupt will not set any bit: the codes for them are both 0. With this code, we cannot determine the exact reason for aborts at runtime. In fact, Intel suggests performance monitoring for deep analysis (see the remainder of this section for details) when programming with TSX, before releasing the software. In addition, Intel provides the `XTEST` instruction to test whether the CPU core is in a transaction region.

We encapsulate the above instructions into C functions in kernel. At the time of Mimosa implementation, we did not find any official support for RTM in the main Linux kernel branch. Although Intel Compiler, Microsoft Visual Studio, and GCC have developed supports for RTM in user programming, they cannot be readily used for kernel programming. Therefore, we refer to Intel Architectures Optimization Reference Manual [39] to emulate RTM intrinsics using inline assembler equivalents. We show the implementation of `_xbegin()` to start the transactional execution of RTM as follows:

```
static __attribute__((__always_inline__)) inline
int _xbegin(void){
    int ret = _XBEGIN_STARTED;
    asm volatile(".byte 0xC7,0xF8; .long 0" :
                 "+a" (ret) :: "memory");
    return ret;
}
```

The default return value is set to `_XBEGIN_STARTED`, which denotes that the transactional execution starts successfully. Next, the transaction starts when `XBEGIN` is executed ("`.byte 0xC7,0xF8`"). The operand "`.long 0`" sets the relative offset of the fallback function address to 0, i.e., the next instruction "**`return`** `ret`". If the transaction starts successfully, the return value is unchanged and returned to callers. Then, the program continues transactional execution until commits successfully. If the transaction is aborted, the program goes to the fallback address (i.e., "**`return`** `ret`"), with the micro-architectural state restored, except that the execution is no longer in transaction and the return value (i.e. the abort status in the `EAX` register) is set properly. Program can decide whether to retry transactional execution again based on the abort status returned in `ret`.

B. The Naïve Implementation

We adopt PolarSSL v1.2.10 as the base of our AES and RSA modules. PolarSSL is a modular and efficient cryptographic library with a very small memory footprint, which is the feature we expect. A smaller work-set means adequate cache resources to complete the transaction. Meanwhile, PolarSSL speeds up the RSA algorithm by employing Chinese remainder theorem (CRT), Montgomery modular exponentiation, and sliding-window exponentiation techniques. It has been adopted by many projects (e.g., LinkSYS, NGINX and OpenVPN) and governmental agencies (e.g., Government of the Netherlands). The AES module is a conventional S-box-based implementation, but we improve it with the AES-NI extension [41].[2] This has three benefits. First, memory footprint is reduced without S-box. Second, performance is boosted with hardware acceleration. More importantly, timing and cache-based [1, 8, 13] side channels of AES implementations are eliminated by running in data-independent time.

In the long-integer module of PolarSSL, a piece of assembly code uses `MMX` registers to accelerate the computation. It is explicitly marked as unfriendly instructions with Intel TSX [40]. Our solution is to replace `MMX` registers with `XMM` registers. This needs only a little modification because both operands are supported in the SSE2 extension.

We implement the steps from **PrCmpt.1** to **PrCmpt.5** described in Section III-B, as a C-language function `mimosa_protected_compute(keyid, in, out)`. It appears to be straightforward to integrate the code in transactional region using the RTM interface: put it after `_xbegin()`, and commit the transaction using `_xend()` that simply invokes `XEND`. As aborts may occur, we invoke `_xbegin()` in an infinite loop, and the execution makes progress if and only if the transaction commits successfully.

```
while (1){
    int status;
    status = _xbegin();
    if (status == _XBEGIN_STARTED)
        break;
}
mimosa_protected_compute(keyid, in, out);
_xend();
```

As mentioned in Section III-B, **PrCmpt.5** erases all sensitive data carefully before committing the transaction, i.e., in `mimosa_protected_compute()`. The sensitive data appear in the following places:

- Allocation buffer: The long-integer module requires dynamically allocated memory.
- Stack of `mimosa_protected_compute()`: The AES round keys and decrypted private keys are stored in the stack of `mimosa_protected_compute()`.
- Register: General purpose registers are involved in computations, and XMM registers are used in AES and long-integer modules.

When we test this naïve implementation, the execution never commits successfully. It is somewhat expected: there are so many restrictions on the execution environment for Intel TSX. In the following, we will demonstrate various causes that lead to aborts and our optimizations. We used the `perf` profiling tool [67] and Intel Software Development Emulator (SDE) version 6.12 [3] for the purpose of discovering abort reasons and performance tuning.

The `perf` profiling tool works with the Intel performance monitoring facility. It supports the precise-event-based sampling (PEBS) function that records the processor state once a particular event happens. In particular, we use the `RTM_RETIRED.ABORTED` event to capture TSX aborts. This event occurs every time a RTM execution is aborted. Based on the dumped processor state, we are able to locate the abort reason and the eventing IP that causes the abort. SDE is the Intel official software emulator for new instruction set extensions. It detects the instructions that are requested to be emulated, and then skips over those instructions and branches to the emulation routines.

C. Performance Tuning

Avoiding Data Conflicts. Both `perf` and SDE reported plenty of data conflicts at first. We found that the modular exponentiation in the naïve implementation used the OS-provided memory allocation library which shares maintained meta data (e.g., free list) for all the threads. As a result, plenty of data conflicts happen when many threads request for new memory simultaneously in multiple cores.

Our solution is that each Mimosa thread monopolizes its own allocation context when in the transactional region. We reserve a static memory buffer as this context for each core. When a Mimosa thread enters the transactional region, it uses the designated context for that core.

We define a global array of allocation contexts. A context is defined for each core as follows:

```
typedef struct{
    unsigned char buffer[MAX_ALLOCATION_SIZE]
                  __attribute__((aligned(64)));
    size_t len;
    size_t current_alloc_size;
    memory_header *first_free;
    ...
    /* other meta data */
} ALLOCATION_CONTEXT;
```

In the transaction, when the memory allocation function is called, the thread first gets its core ID and uses it to locate its allocation context. Then it performs actual memory allocation in this context as follows.

```
void *mimosa_malloc(size_t len){
    ALLOCATION_CONTEXT *context;
    int id;
    id = smp_processor_id();
    context = allocation_context + id;
    ...
    /* Actual allocation in the context */
}
```

The first member in `ALLOCATION_CONTEXT` is aligned on a 64-byte boundary (cache line size), which is the granularity to track the read/write-set addresses. This prevents false sharing between continuous contexts. False sharing happens when two threads access their distinct memory locations in the same cache line, thus would cause data conflict unexpectedly.

[2] Newer versions of PolarSSL also support AES-NI. We develop it independently to avoid using shared memories.

With this tuning, Mimosa can work very well on SDE.[3] We configured the CPU parameters in SDE so that the cache size is identical to Intel Core i7 4770S (our target CPU), and 8 Mimosa threads run without abort during extensive experiments on SDE. This proves that our implementation is fully compatible with the Intel TSX specification and no data conflict is caused by Mimosa itself.

Disabling Interrupts and Preemption. However, SDE does not simulate real-time interrupts to support multi-tasking. The private key computation is time-consuming. Therefore, it is very likely that transactional execution is interrupted by task scheduling on real hardware, which definitely causes aborts. Other interrupts may also cause aborts. To give Mimosa enough time to complete computations, interrupts and kernel preemption are temporarily disabled when in transactional region. All CPU-bound encryptions including TRESOR [56], PRIME [29] and Copker [30], require disabling interrupts to ensure *atomicity*, while Mimosa requires it for *efficiency* because Intel TSX itself ensures atomicity already.

Delay after Continuous Aborts. At this point, the *abort cycle ratio*[4] is relatively high, resulting in bad performance. The perf profiling tool is unable to provide obvious information about abort reasons. The eventing IPs recorded by PEBS spread across the transactional region. Meanwhile, most of reported reason codes are ABORTED_MISC5, which has a very ambiguous description by Intel. We list the abort reason descriptions as follows.

- ABORTED_MISC1: Memory events, e.g., read/write capacity and conflicts.
- ABORTED_MISC2: Uncommon conditions.
- ABORTED_MISC3: Unfriendly instructions.
- ABORTED_MISC4: Incompatible memory type.
- ABORTED_MISC5: Others, none of the previous four categories, e.g., interrupts.

At first, we suspect that the aborts would be caused by non-maskable interrupt (NMI); however, this explanation is immediately ruled out after examining the NMI counter through the /proc/interrupts interface. We have contacted Intel for support.[5] As stated before, Intel provides no guarantees as to whether transactional execution will successfully commit and there are numerous implementation specific reasons that may cause aborts [40].

We notice that Intel recommends a delay before retrying if the abort was caused by data conflict [39]. Although we encounter a different cause, we still modify Mimosa to force a short delay after several failed transactions. As a result, the success rate is significantly improved. The number of tries before a delay is 5, which is an empirical value suggested in [84] and also verified in our experiments. After extensive experiments balancing the throughput under the single-threaded and the multi-threaded scenarios, we have identified 10 clock ticks as an optimized value for this delay.[6]

```
while(!success){
    int times = 0;
    /* Disable interrupts and preemption */
    get_cpu();
    local_irq_save(flags);
#ifdef TSX_ENABLE /* Switch of Mimosa_NO_TSX */
    while(1){
        int status;
        if(++times == MAX_TRIES)
            goto delay;
        status = _xbegin();
        if(status == _XBEGIN_STARTED)
            break;
    }
#endif
    mimosa_protected_compute(keyid, in, out);
    success = 1;
#ifdef TSX_ENABLE
    _xend();
#endif
delay:
    /* Enable interrupts and preemption */
    local_irq_restore(flags);
    put_cpu();
    if(!success){
        /* Delay after several aborts */
        set_current_state(TASK_INTERRUPTIBLE);
        schedule_timeout(10);
    }
}
```

Figure 2: Code Snippet in Mimosa

After these tunings, almost all the remaining aborts occur at the very beginning of the transactions. Therefore, although we are unable to identify the exact reason(s) of the remaining aborts, or to avoid all aborts, they only waste a very small number of CPU cycles. The abort cycle ratio is low, as more than 95% of the CPU cycles are used in successful transactions. Figure 2 shows the final code snippet.

We would like to point out that the significant performance improvement is the result of all the tuning approaches. It might appear that the abort issue is solved by the last attempt (i.e., adding delays); however, it wouldn't be successful if we skip any of the previous steps. Our perception is that we may not be able to completely avoid all the aborts eventually: (1) the simulation results have shown that our implementation is correct; (2) the Intel official tools

[3] Mimosa is modified slightly to conform to SDE that runs in the user space.

[4] The number of CPU cycles in the aborted transactions divided by the total number of CPU cycles in all transactions.

[5] Until the submission of this manuscript, we have not received any reply.

[6] For the HTM feature in zEC12 systems, IBM also suggests a random delay before retrying a transaction on aborts [45]; and the optimal delay depends on the particular abort reason, the CPU design and configuration.

are unable to identify or provide the details of the aborts; and (3) Intel TSX and the speculative nature of transactional memory do not guarantee all correctly implemented transactions to commit, e.g., cache coherence traffic may at times appear as conflicting requests and may cause aborts.

D. Utility Issues

Based on the design described in Section III, Mimosa needs an off-line machine to securely generate an encrypted RSA private key file, and a to-be-protected machine that runs the Mimosa service. The preparation utility in the off-line machine generates RSA key pairs and encrypts them by an AES master key derived from the same user password. The encrypted key file is then copied to the to-be-protected machine.

Mimosa provides private-key computation services to user space through the `ioctl` system call. Based on the command code and key ID, Mimosa outputs the public key of the corresponding key pair, or performs a private-key operation on the input data and outputs the result. In addition, the `ioctl` interface is encapsulated into an OpenSSL engine. A RSA key is selected via `ENGINE_load_private_key()` and then can be used in an OpenSSL-compatible way. We use this API to integrate Mimosa into the Apache HTTPS server in the evaluation.

E. Applicability

Although the Mimosa prototype is implemented on Intel Haswell CPUs using the RTM interface of TSX,[7] our solution is applicable to other platforms. Firstly, it can be implemented using the HLE interface. In particular, if the protected computing is executed as a transaction using HLE, `XTEST` will be used to determine whether it is in transactional execution or not. If it is in normal execution (i.e., the transaction aborts for some reasons), the protected computing will not continue and the transactional execution will be retried.

Most HTM solutions share a similar programming interface. We will show that, in other HTM implementations, the counterparts of the Intel TSX `XBEGIN` and `XEND` instructions can be easily identified, and the abort processing conforms to the Mimosa design. For example, in the HTM facility of IBM zEC12 systems [45], transactions are defined by the instructions `TBEGIN` and `TEND`. On abort, the `PC` register is restored to the instruction immediately after `TBEGIN`, and a condition code is set to a non-zero value. Typically, a program tests the condition code after `TBEGIN` to either start the transaction execution if `CC=0` or branch to the program-specified fallback function (i.e., retry or delay in Mimosa) if `CC!=0`. AMD has proposed its own HTM extension since 2009, called Advanced Synchronization Facility (ASF), but currently the products are not ready. Based on the public specifications [2], ASF provides similar instructions to specify a transactional region (i.e., `SPECULATE` and `COMMIT`) and also tracks memory accesses in CPU caches. The transaction starts after the execution of `SPECULATE` and commits at `COMMIT`. An instruction following `SPECULATE` checks the status code and jumps to the program-specified fallback handler if it is not zero. ASF has a slightly different programming interface in that all the to-be-tracked memories for atomic access must be explicitly specified using declarator instructions (i.e., the `LOCK` prefix).

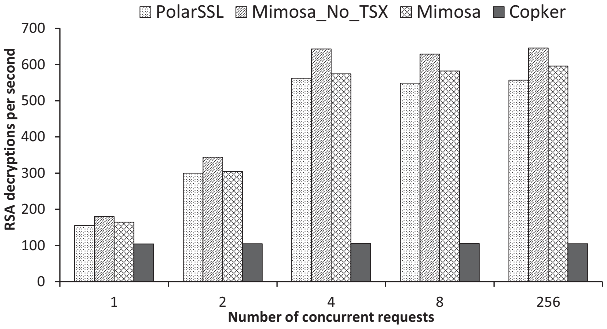

Figure 3: Local Performance

Finally, most existing HTM implementations use on-chip caches or store buffers [2, 27, 33, 45, 82] for the transaction execution, so they can also work with Mimosa to prevent cold-boot attacks.

V. Performance Evaluation

This section presents the experimental results by measuring the performance of Mimosa. We carried out experiments on a machine with an Intel Core i7 4770S CPU (4 cores), running a patched Linux Kernel version 3.13.1. In these experiments, we compared Mimosa with: (1) the official PolarSSL version 1.2.10 with default configurations, (2) Mimosa_No_TSX, which is the same as Mimosa but not in transactional execution, by turning off the `TSX_ENABLE` switch (see Figure 2), and (3) Copker[8] [30]. We used 2048-bit RSA keys in the first three experiments.

A. Local Performance

First, Mimosa ran as a local RSA decryption service, and we measured the maximum number of decryption operations per second. We evaluated Mimosa's performance at different concurrency levels, and compared it with other approaches, as introduced above. As shown in Figure 3, all

[7] In August 2014, Intel announced a bug in the released TSX implementation, and suggested disabling TSX on the affected CPUs via a microcode update [42]. During our experiments, the Mimosa prototype works well as described in Section V. Note that TSX is still supported in newer CPUs, e.g., Intel Core M-5Y71 CPU launched in Q4 2014 [43].

[8] The authors of Copker provided us with their source code, and we revised it slightly to work for Intel Core i7 4770S CPU.

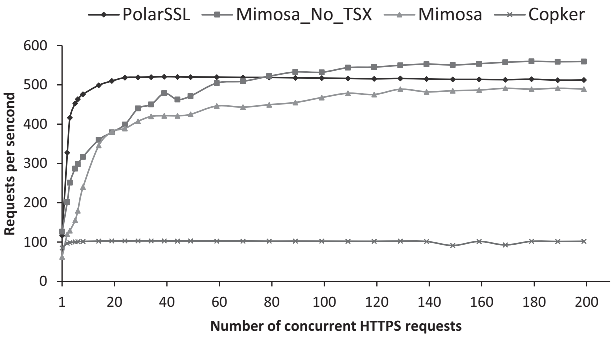

Figure 4: HTTPS Throughput

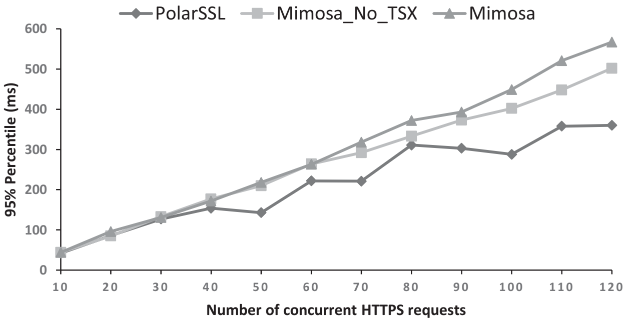

Figure 5: 95% Percentile Latencies of HTTPS

the approaches exhibit similar performance except Copker, which can use only one core due to a shared L3 cache in this Intel Core i7 4770S processor: the Copker core works in the `write-back` cache-filling mode, and forces all other cores to the `no-fill` mode. With a shared L3 cache, only one Copker task works, while all other tasks have to wait. Under all concurrency levels, the abort cycle ratio of Mimosa is always under 5%. Mimosa and Mimosa_No_TSX performs even better than PolarSSL. It is probably because PolarSSL is subject to task scheduling, while kernel preemption is disabled in Mimosa and Mimosa_No_TSX.

B. *HTTPS Throughput and Latency*

We evaluated Mimosa in a more practical setting. In this experiment, Mimosa, Mimosa_No_TSX and PolarSSL served as the RSA decryption module in the Apache HTTPS server, respectively, and then we measured the throughput and latency. The web page used in the experiment was 4 KBytes in size. The server and the client were located in an isolated Gigabit Ethernet.

The client ran an ApacheBench process that issued 10,000 requests at different concurrence levels, and the number of HTTPS requests handled per second was shown in Figure 4. For Mimosa, the maximum throughput loses 17.6% of its local capacity. Mimosa_No_TSX loses 13.5%, and PolarSSL loses 6.5%. From the results, we can estimate that the first 6.5% loss (for all approaches) should be attributed to the unavoidable overhead of SSL and network transmission. Disabling kernel preemption has a negative impact on concurrent tasks, so Mimosa perform worse than the user-space PolarSSL; but Mimosa_No_TSX performs still a little better than PolarSSL after the number of concurrent requests reaches 80. The additional loss of capacity in Mimosa shall be caused by aborted cycles.

We measured the HTTPS latency using curl (one client, disabling the keep-alive option). The average SSL handshake times were 9.98ms, 9.04ms and 10.94ms, when PolarSSL, Mimosa_No_TSX and Mimosa served in the HTTPS server, respectively. We also stressed the HTTPS server with different loads to measure its 95th percentile using ApacheBench. The total issued HTTPS request is 10,000. As shown in Figure 5, the negative impact of disabling kernel preemption and aborted cycles in Mimosa is acceptable under medium loads. The 95th percentile latency of Mimosa is about 1.6 times that of PolarSSL.

C. *Impact on Concurrent Processes*

We used the Geekbench 3 benchmark [66] to evaluate how Mimosa influenced other concurrent applications. Running concurrently with the RSA computations by each evaluated solution, Geekbench 3 measures the machine's integer, floating point and memory performance, i.e., the computation capacity remained for concurrent processes. The Geekbench 3 scores for both the single-core mode and the multi-core mode are shown in Figure 6. The baseline score was measured in a clean environment without any process except Geekbench 3, indicating the machine's full capacity. Others were measured when the benchmark was running concurrently with the Apache HTTPS service at the workload of 80 requests per second. Note that we would like to ensure that all the evaluated approaches worked at the same RSA computation workload. Since the maximum throughput of Copker is around 100 HTTPS requests per second (Figure 4), we pick 80 requests per second in this experiment.

In Figure 6, the `integer`, `floating point` and `memory` scores denote the integer instruction performance, floating point instruction performance and memory bandwidth, respectively. Overall, the single-core scores of PolarSSL, Mimosa and Mimosa_No_TSX are very close, except Copker. When Geekbench 3 occupies more cores, the overhead for handling the HTTPS requests becomes nontrivial – there is a clear gap between the baseline scores and others. User-space approaches (i.e., PolarSSL) introduce a little less impact on concurrent processes than kernel space approaches (i.e., Mimosa and Mimosa_No_TSX) where kernel preemption is disabled. In Figure 3, we find that preemption-disabled approaches are more efficient because more resources are occupied by them. However, this also means that concurrent processes cannot be served in time,

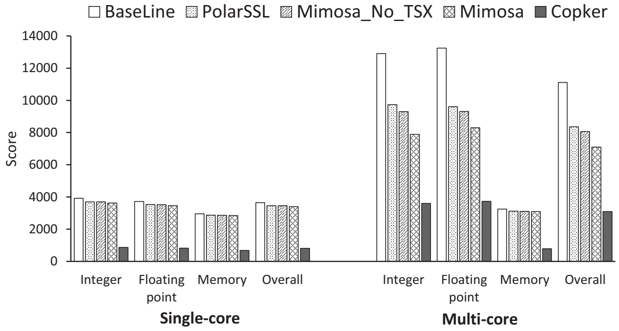

Figure 6: Geekbench 3 Scores under RSA Computations

as shown in Figure 6. Meanwhile, Mimosa_No_TSX introduces less overhead than Mimosa, because Mimosa has to waste some aborted cycles. Last, we see a significant drop in Copker for both the single-core mode and the multi-core mode, because all other CPU cores are forced to enter the `no-fill` cache-filling mode when Copker is running. That is, the benchmark runs without caches at times.

D. Scalability

Finally, we evaluated the performance of Mimosa with growing RSA key lengths, to prove its potential applicability to other cryptographic algorithms requiring more memory and heavier computation. In this experiment, Mimosa ran locally to accomplish the maximum performance (decryption operations per second). We can see from Table I that as the key length grows, the performance of Mimosa decreases at a similar pace with Mimosa_No_TSX. This indicates that the size of L1D caches has not become the bottleneck to support stronger keys for up to 4096-bit RSA.

We measured the size of dynamically allocated memory in a transaction which approximates the whole work set for 4096-bit RSA computations. It turned out that the allocated memory size was about 9.3 KBytes, which is far less than the supported write-set size of Intel TSX evaluated in [52], 26 KBytes. This proves that there is still great potential for supporting other memory-hungry algorithms.

VI. Security Analysis

The security goals presented in Section III-A are achieved with Intel TSX as follows. Any attack that attempts to access the private keys and sensitive intermediate states during the protected computing phase automatically triggers the *hardware abort handler*, clearing all sensitive information. If it commits successfully, the transactional execution is guaranteed to be performed within L1D caches and always ends with clearing all sensitive data.

In this section, we introduce experimental validations to verify that Mimosa has achieved these goals. Then, the remaining attack surfaces are discussed, and we compare Mimosa with other defenses against cold-boot attacks (and also other memory disclosure attacks) on private keys.

Table I: Local Performance for Different Key Lengths

Key Length (bits)	1024	2048	3072	4096
Mimosa (decryptions/sec)	3726	596	199	76
Mimosa_No_TSX (decryptions/sec)	3798	646	214	95
#(Mimosa)/#(Mimosa_No_TSX)	98%	92%	93%	80%

A. Validation

To validate that software memory disclosure attacks cannot obtain the sensitive information of Mimosa, we implemented a privileged "attack" program (the validator), which actively reads the memory addresses used in Mimosa through the `/dev/mem` interface. These memory locations are fixed once Mimosa has been loaded. `/dev/mem` is a virtual device that provides an interface to access the system's physical memory, if kernel is configured with `CONFIG_STRICT_DEVMEM` disabled. Specifically, every second, the validator read the global array that stores the plaintext private keys in Mimosa. We kept the validator running for more than 5 hours (approximately 20,000 reads), while there were 256 threads repeatedly calling the Mimosa services at the full speed. Throughout the experiment, the validator returned cleared data only. That is, the attacks are unable to read any sensitive information in Mimosa. Note that our "attack" program is much more powerful than the real-world software memory disclosure attacks, because this program runs with the root privilege and knows the exact memory address for sensitive data. As a comparison, when we disabled the TSX protection (i.e., Mimosa_No_TSX), almost every access obtained the plaintext private keys.

We also used `Kdump` to dump the kernel memory to find any suspicious occurrence of sensitive data. `Kdump` allows a dump-capture kernel to take a dump of the system kernel's physical memory and the register state of other cores when the system crashes. Note that this mechanism sends *non-maskable* interprocessor interrupts (IPIs) to other CPU cores to halt the system.[9] We ran Mimosa intensively. Meanwhile, we crashed the system by writing 'c' to `/proc/sysrq-trigger`. After dumping the system kernel to the disk, we searched for the RSA private key and the AES master key in the file. The AES key has two forms: the original 128-bit key and 10 rounds key schedule. First, for the AES key schedule, we used the AESKeyFinder tool [36] to analyze the captured image. As a result, we found no matching key schedule of the one used in Mimosa. Meanwhile, for the original AES key and the RSA private

[9] NMIs destroy atomicity since only local interrupt is disabled. This is the reason that CPU-bound solutions like Amnesia and Copker suggest modifying NMI handlers to immediately clear all sensitive keys [30, 73]. Technically, TRESOR and PRIME are subject to similar attacks. However, in Mimosa, the private key and intermediate states are automatically cleared once NMIs happen, eliminating the need to modify NMI handlers.

key, we used the bit string matching program bgrep to search for (pieces of) the known keys. The image had no matching of both the AES and RSA keys (including p, q, d and other CRT elements). We have never found a binary sequence that overlaps for more than 3 bytes with any key. On the contrary, when the same experiment was conducted on Mimosa_No_TSX, we got a great deal of copies of both the AES master key and the private keys. They came from three sources. First, Kdump dumped the register states of all the cores when the system crashed. Second, interrupted threads left the decrypted keys uncleared. Third, the process control blocks of the threads contained the register states as a result of context switching.

We did not launch a cold-boot attack or probe the bus to validate that these is no data leakage to the RAM chips. However, according to Intel Architectures Optimization Reference Manual [39] (Section 12.1.1), when cache eviction in the write-set happens, transaction will abort immediately, thus modified data are discarded inside L1D caches. Note that the plaintext private keys and other sensitive intermediate states are in the write-set, because they are generated only after the transaction starts. Therefore, sensitive data would appear nowhere other than L1D caches. We would also like to argue that this character is a necessary requirement to correctly implement Intel TSX. If a modified cache line is evicted to some place outside the boundary of Intel TSX, its value will be available to other components – an obvious contradiction to the nature of transactional memory.

B. *Remaining Attack Surface*

Mimosa relies on Intel TSX to ensure the confidentiality of the protected RSA private key in the protected computing phases. However, the CPU cache which is the base of TSX is constrained in size and shared among all cores. This might lead to denial of service (DoS) attacks. Most Intel CPUs implement 8-way set associative L1D cache, so 9 memory addresses in the write-set mapping to the same cache set will abort the transaction. Moreover, Intel does not guarantee all cache lines of a cache set can be attributed to the transactional execution. Besides, a process with very intensive memory accesses may halt the Mimosa service too, because there is a big chance that this process will evict the cache lines that Mimosa is occupying due to shared L3 caches.

We evaluated how seriously a memory-intensive program would impact Mimosa, by launching the memory test of the Geekbench 3 benchmark concurrently with Mimosa. In this experiment, 4 kinds of STREAM workloads were performed on all CPU cores, resulting in 10.3 GBytes memory transferred per second. The machine (clean environment) supports a maximum transfer rate of 13.7GB/s. The average performance of Mimosa fell to 137 decryptions per second. That is a degradation of 77.0%, compared with the original result of 596 per second in Table I. Meanwhile, performance of Mimosa_No_TSX degraded 42.0% in the same experiment; and the degradation of original PolarSSL is 44.8%. Therefore, only about 30% of the performance degradation was caused by the use of TSX in Mimosa and aborts due to intensive memory access, and about 40% was by the resource occupation of STREAM. We also measured the abort cycle ratio in Mimosa – it raised to 52% under the "DoS attacks", compared with less than 5% in normal settings (see Section V-A). We have tried other different memory-intensive programs, and all of them have less performance impact. Note that, even in this under-attack case, Mimosa has performance advantage over the existing CPU-bound solutions (PRIME and Copker, see Table II).

Attackers might attempt to exploit side channels to compromise private keys. Cache-based side channels [8, 13] do not exist in Mimosa, because AES-NI is free of timing attacks [41] and the RSA computations are performed entirely in caches. Other timing attacks [6, 15] against PolarSSL used in the current Mimosa implementation, can be prevented by RSA-blinding [15], which will be in our future work.

Implemented as a kernel module, Mimosa needs to assume the integrity of OS kernel, so integrity protections are required to work complementarily. While the kernel integrity solutions protect the Mimosa service binaries from being modified, Mimosa prevents the memory disclosure attacks that do not violate the integrity of executable binaries. Most existing OS integrity solutions are based on virtualization such as SecVisor [70], SBCFI [64], OSck [38] and Lares [63], while kGuard [47] augments the kernel with inline guards. Integrating Mimosa with these solutions will be also included in our future work.

TRESOR-HUNT [11] exhibits an advanced DMA attack that injects malicious codes into OS kernel (i.e., breaks the integrity of kernel) and then accesses the AES key in debug registers. Fortunately, this attack can be countered by configuring IOMMU [76], monitoring bus activities [75], or leveraging the system management mode (SMM) [85]. Laser scan, another physical memory disclosure attack, was proposed to read information in power-on smart cards [68]. But this method needs to depackage the chip to remove metal layers, and requires the chip be halted in the target state (i.e., no update operation during the scan). So it is not a threaten of Mimosa.

C. *Comparison*

Currently, there are three implementations of asymmetric cryptosystems that are resistant to cold-boot attacks, namely, PRIME [29], Copker [30] and the proposed work. These solutions adopt the same key-encryption-key structure, in which an AES master key is stored in privileged CPU registers throughout the operation of the system. The private key is decrypted on demand to perform RSA decryption or signing. Table II summaries the properties of three approaches in terms of OS security assumption, efficiency and private-key

Table II: Comparison of the RSA Implementations against Cold-boot Attacks

Solution	OS Security Assumption		Efficiency	Private-Key Computation
	Master Key Protection	Private Key Protection	Compared with PolarSSL	Implementation Language
Mimosa	X+D	X	Comparable	C
PRIME	X+D	X+A	Approximately 1/9†	Assembly
Copker	X+D	X+A+R	Approximately 1/4‡	C

X: The integrity of executable binary in kernel.
D: No illegal access to debug registers.
A: Atomicity guarantee of private-key computations.
R: No illegal memory read operation.
†: It is drawn directly from [29].
‡: It is about the number of separate cache sets divided by that of cores [30]. Intel Core i7 has 4 cores with shared L3 caches.

computation implementation. Hardware assumptions are not shown in this Table, such as Intel TSX, cache-filling modes, CPU privilege rings, etc.. Firstly, TRESOR [56] is used in all three solutions to protect the AES master key, the security of which depends on the integrity of the kernel executable without interfaces to debug registers. However, the private key protections are very different:

- PRIME uses AVX registers to store private keys, and requires atomicity guarantee of private-key computations; otherwise, the unprivileged AVX registers may be accessed by attackers who interrupts the computations.
- Copker depends on both process isolation from OS as well as atomicity guarantee of private-key computations, because illegal memory read operations will synchronize the private key to RAM chips when it is decrypting or signing messages.
- Mimosa only assumes the kernel integrity, and its atomicity is guaranteed by hardware but not OS.

Secondly, with the hardware support from Intel TSX, Mimosa significantly outperforms PRIME and Copker. Finally, because the private-key computation is implemented in C-language, it is easier for Mimosa and Copker to support other asymmetric algorithms such as DSA and ECDSA.

VII. Related Work

A. Attack and Defense on Cryptographic Keys

As verified by the experiments [35], more copies of cryptographic keys in memory result in a greater leakage risk. Several works have been done to minimize the occurrence of sensitive data. Secure deallocation [18] erases data either on deallocation or within a short and predictable period, reducing the copies of sensitive data in unallocated memory. Harrison *et al.* provide ways to keep only one copy of cryptographic keys in allocated memory [35]. In [61], a 1024-bit RSA private key is scrambled and dispersed in memory, but re-assembled in x86 SSE XMM registers when computations are needed, to achieve no copy of private keys in memory. To avoid sensitive data being leaked to disks, Scrash [14] removes sensitive data from crash reports in the case of program failures. Mimosa follows the same spirit that the sensitive cryptographic keys shall appear in minimal locations and minimal periods, and we employ HTM to enforce this principle, ensuring that the private keys are cleared after computations and appear only in CPU caches. More importantly, Mimosa reactively clears the sensitive data whenever an attack attempt is detected.

AESSE [55], TRESOR [56] and Amnesia [73] are proposed to prevent the cold-boot attack [32] on full-disk encryption, by storing AES keys in registers only. This CPU-bound approach is extended to the asymmetric cryptographic algorithms that require much more storages. Using the AES key protected by TRESOR as a key-encryption key, PRIME [29] implemented the RSA computations in AVX registers while Copker [30] did it in caches. Mimosa implements the RSA algorithm against cold-boot attacks, but provides much better performance than PRIME and Copker.

The register-based AES implementations such as TRESOR can also prevent read-only DMA attacks [7, 12, 76] that passively read from memory. Advanced DMA attacks [11] can exploit malicious firmware to actively write injected codes into the memory space of OS kernel, and then access the keys in registers. This advanced DMA attack can be detected or restricted by configuring IOMMU [76], monitoring bus activities [75], leveraging SMM [85], or the timed challenge-response protocol [51].

PixelVault uses GPUs as the secure container for cryptographic computing [81]. All sensitive data and executable binaries are loaded into the caches and registers of GPUs in the initialization phase, so (malicious) binaries on CPUs cannot access these data and binaries on GPUs. Therefore, it does not require the integrity of OS kernel except during initialization. PixelVault requires GPUs dedicated for cryptographic computing; however, Mimosa dynamically builds secure containers within CPUs, and releases resources when not in-use.

Intel Software Guard eXtensions (SGX) [44] plan to provide a hardware-enabled secure container that is isolated from other processes. Confidentiality and integrity of the protected process will be maintained even in the presence of privileged malware. Although SGX is not commercially ready, it shows the same tendency and potential as TSX that secure systems can be built on top of hardware features.

There are cryptographic methods that mitigate the attacks on cryptographic keys. Threshold cryptography [26] splits the key into multiple parts to withstand partial disclosure.

Leakage-resilient cryptography [4, 5] is secure against memory attacks where the adversary measures a large fraction of the bits of keys. However, these solutions become useless if the attackers compromise all bits of the key step by step. In order to prevent cross-VM side-channel attacks [86], Hermes [62] applied threshold cryptography to build a cryptographic system on several virtual machines in the cloud, each of which hold a partition of the private key. The key is re-shared periodically in Hermes to invalidate the compromised partitions. White-box cryptography [19, 20] plants a fixed key into the algorithm implementation. Even if an attacker obtains the entire memory, he or she cannot find out the key. However, white-box cryptography has efficiency limits and is not applicable to asymmetric cryptographic algorithm so far.

B. Transactional Memory Implementation

Various transactional memory solutions have been proposed, from hardware-based solutions [2, 27, 33, 40, 45, 54, 82] to software transactional memory (STM) [16, 34, 60, 71] and hybrid schemes [25, 49]. HTM pushes transactional memory primitives into hardware, hence minimizes performance overheads greatly.

HTM usually temporarily stores updated memory in CPU-bound caches or store buffers [2, 27, 33, 45, 82] before commit, and discards the modified data on aborts. Another hardware solution, LogTM [54] updates memory directly and saves the unmodified value in a per-thread memory log; on aborts, state is restored by inspecting through the logs.

C. Transactional Memory Application

Transactional memory boosts thread-level parallelism, and is applied in different services such as game servers [53, 87] and database systems [46] to improve performance.

By maintaining security-relevant shared resources in the read/write-sets of Haskell STM [34], TMI enforces authorization policies whenever such a resource is accessed [9, 10]. TMI and Mimosa depend on transactional memory to inspect all accesses to sensitive resources. TMI enforces authorization on every access, while Mimosa ensures confidentiality by clearing sensitive keys once any illegal read operation occurs.

TxIntro [52] is another application of Intel TSX. It leverages the strong atomicity of HTM to synchronize virtual machine introspection (VMI) and guest OS execution, so that VMI is performed more timely and consistently. TxIntro and Mimosa use Intel TSX in very different ways. TxIntro monitors the read-set to detect concurrent update operations that cause inconsistence, while Mimosa monitors the write-set to detect illegal concurrent read operations.

Transactional memory improves the multi-thread support in dynamic binary translation to guarantee correct execution of concurrent threads [21]. In addition, the abstraction of transactional memory is extended to kernel extensions [69] and code functions [72], and these transaction semantics are used to recover a system from failures.

VIII. Conclusion

We present Mimosa, an implementation of the RSA cryptosystem with substantially improved security guarantees on the private keys. With the help of HTM, Mimosa ensures that only Mimosa itself is able to access plaintext private keys in a private-key computation transaction. Any unauthorized access would automatically trigger a transaction abort, which immediately clears all sensitive data and terminates the cryptographic computations. This thwarts software memory disclosure attacks that exploit vulnerabilities to stealthily read sensitive data from memory without breaking the integrity of executable binaries. Meanwhile, the whole protected computing environment is constrained in CPU caches, so Mimosa is immune to cold-boot attacks on RAM chips.

We implemented the Mimosa prototype with Intel TSX. We have simulated the most powerful software memory disclosure “attacks” and validated that unauthorized access to sensitive data could only obtain erased or encrypted copies of private keys. Kernel dump when Mimosa is running fails to capture any sensitive content, either. Moreover, performance evaluation shows that Mimosa exhibits comparable efficiency with conventional RSA implementations, which do not provide the mentioned security guarantees. We also demonstrate that Mimosa serves well in the real-world applications, e.g., HTTPS service.

Acknowledgement

Le Guan, Jingqiang Lin, Jiwu Jing and Jing Wang were partially supported by National 973 Program of China under award No. 2013CB338001 and Strategy Pilot Project of Chinese Academy of Sciences under award No. XDA06010702. Bo Luo was partially supported by NSF OIA-1028098, NSF CNS-1422206 and University of Kansas GRF-2301876. The authors would like to thank Dr. Yuan Zhao at Institute of Information Engineering, Chinese Academy of Sciences, Prof. Peng Liu at Pennsylvania State University and the anonymous reviewers for their valuable comments.

References

[1] O. Aciicmez, W. Schindler, and C. Koc, “Cache based remote timing attack on the AES,” in *Topics in Cryptology - CT-RSA*, 2007, pp. 271–286.

[2] Advanced Micro Devices, “Advanced Synchronization Facility, proposed architectural specification (revision 2.1),” 2009, http://developer.amd.com/wordpress/media/2013/02/45432-ASF_Spec_2.1.pdf.

[3] Ady Tal, “Intel Software Development Emulator,” http://software.intel.com/en-us/articles/intel-software-development-emulator.

[4] A. Akavia, S. Goldwasser, and V. Vaikuntanathan, “Simultaneous hardcore bits and cryptography against

memory attacks," in *6th Theory of Cryptography Conference (TCC)*, 2009, pp. 474–495.

[5] J. Alwen, Y. Dodis, and D. Wichs, "Survey: Leakage resilience and the bounded retrieval model," in *4th International Conference on Information Theoretic Security (ICITS)*, 2010, pp. 1–18.

[6] C. Arnaud and P.-A. Fouque, "Timing attack against protected RSA-CRT implementation used in PolarSSL," in *Topics in Cryptology - CT-RSA*, 2013, pp. 18–33.

[7] M. Becher, M. Dornseif, and C. Klein, "Firewire: All your memory are belong to us," in *6th Annual CanSecWest Conference*, 2005.

[8] D. Bernstein, "Cache-timing attacks on AES," 2004, http://cr.yp.to/antiforgery/cachetiming-20041111.pdf.

[9] A. Birgisson, M. Dhawan, U. Erlingsson *et al.*, "Enforcing authorization policies using transactional memory introspection," in *15th ACM Conference on Computer and Communications Security (CCS)*, 2008, pp. 223–234.

[10] A. Birgisson and U. Erlingsson, "An implementation and semantics for transactional memory introspection in Haskell," in *4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009, pp. 87–99.

[11] E.-O. Blass and W. Robertson, "TRESOR-HUNT: Attacking CPU-bound encryption," in *28th Annual Computer Security Applications Conference (ACSAC)*, 2012, pp. 71–78.

[12] B. Bock, "Firewire-based physical security attacks on Windows 7, EFS and BitLocker," 2009.

[13] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *8th Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2006, pp. 201–215.

[14] P. Broadwell, M. Harren, and N. Sastry, "Scrash: A system for generating secure crash information," in *12th USENIX Security Symposium*, 2003.

[15] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[16] B. Carlstrom, A. McDonald, H. Chafi *et al.*, "The Atomos transactional programming language," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006, pp. 1–13.

[17] J. Chow, B. Pfaff, T. Garfinkel *et al.*, "Understanding data lifetime via whole system simulation," in *13th USENIX Security Symposium*, 2004.

[18] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," in *14th USENIX Security Symposium*, 2005.

[19] S. Chow, P. Eisen, H. Johnson, and P. van Oorschot, "A white-box DES implementation for DRM applications," in *2nd ACM Workshop on Digital Rights Management (DRM)*, 2002, pp. 1–15.

[20] ——, "White-box cryptography and an AES implementation," in *9th International Workshop on Selected Areas in Cryptography (SAC)*, 2002, pp. 250–270.

[21] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis, "Thread-safe dynamic binary translation using transactional memory," in *14th IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2008, pp. 279–289.

[22] C. Cowan, C. Pu, D. Maier *et al.*, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *7th USENIX Security Symposium*, 1998.

[23] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuardTM: Protecting pointers from buffer overflow vulnerabilities," in *12th USENIX Security Symposium*, 2003.

[24] CVE Details, "Linux kernel vulnerability statistics," http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html, accessed in November 2014.

[25] P. Damron, A. Fedorova, Y. Lev *et al.*, "Hybrid transactional memory," in *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 336–346.

[26] Y. Desmedt and Y. Frankel, "Threshold cryptosystems," in *Advances in Cryptology - CRYPTO*, 1989, pp. 307–315.

[27] D. Dice, Y. Lev, M. Moir *et al.*, "Early experience with a commercial hardware transactional memory implementation," in *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 157–168.

[28] A. Dunn, M. Lee, S. Jana *et al.*, "Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels," in *10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 61–75.

[29] B. Garmany and T. Müller, "PRIME: Private RSA infrastructure for memory-less encryption," in *29th Annual Computer Security Applications Conference (ACSAC)*, 2013.

[30] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with private keys without RAM," in *21st ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[31] G. Guninski, "Linux kernel 2.6 fun, Windoze is a joke," 2005, http://www.guninski.com/where_do_you_want_billg_to_go_today_3.html.

[32] J. Halderman, S. Schoen, N. Heninger *et al.*, "Lest we remember: Cold boot attacks on encryption keys," in *17th USENIX Security Symposium*, 2008.

[33] L. Hammond, V. Wong, M. Chen *et al.*, "Transactional memory coherence and consistency," in *ACM*

SIGARCH Computer Architecture News, vol. 32, 2004, p. 102.

[34] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005, pp. 48–60.

[35] K. Harrison and S. Xu, "Protecting cryptographic keys from memory disclosure attacks," in *37th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 137–143.

[36] N. Heninger and A. Feldman, "AESKeyFinder," https://citp.princeton.edu/research/memory/code/.

[37] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *20th International Symposium on Computer Architecture (ISCA)*, 1993, pp. 289–300.

[38] O. Hofmann, A. Dunn, S. Kim *et al.*, "Ensuring operating system kernel integrity with OSck," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 279–290.

[39] Intel Corporation, "Intel 64 and IA-32 architectures optimization reference manual."

[40] ——, "Chapter 8: Intel transactional memory synchronization extensions," in *Intel Architecture Instruction Set Extensions Programming Reference*, 2012.

[41] ——, "Intel advanced encryption standard (AES) new instructions set (revision 3.01)," 2012.

[42] ——, "Desktop 4th generation Intel Core processor family, specification update," 2014.

[43] ——, "Intel Core M-5Y71 processor," 2014, http://ark.intel.com/products/84672/Intel-Core-M-5Y71-Processor-4M-Cache-up-to-2_90-GHz.

[44] ——, "Intel Software Guard Extensions Programming Reference," 2014, https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[45] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for IBM System Z," in *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 25–36.

[46] T. Karnagel, R. Dementiev, R. Rajwar *et al.*, "Improving in-memory database index performance with Intel transactional synchronization extensions," in *20th IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 476–487.

[47] V. Kemerlis, G. Portokalidis, and A. Keromyti, "kGuard: Lightweight kernel protection against return-to-user attacks," in *21st USENIX Security Symposium*, 2012.

[48] V. Kolontsov, "WU-FTPD core dump vulnerability (the old patch doesn't work)," 1997, http://insecure.org/sploits/ftpd.pasv.html.

[49] S. Kumar, M. Chu, C. Hughes *et al.*, "Hybrid transactional memory," in *11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006, pp. 209–220.

[50] M. Lafon and R. Francoise, "CAN-2005-0400: Information leak in the Linux kernel ext2 implementation," 2005, http://www.securiteam.com.

[51] Y. Li, J. McCune, and A. Perrig, "VIPER: Verifying the integrity of peripherals' firmware," in *18th ACM Conference on Computer and Communications Security (CCS)*, 2011, pp. 3–16.

[52] Y. Liu, Y. Xia, H. Guan *et al.*, "Concurrent and consistent virtual machine introspection with hardware transactional memory," in *20th IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 416–427.

[53] D. Lupei, B. Simion, D. Pinto *et al.*, "Transactional memory support for scalable and transparent parallelization of multiplayer games," in *5th European Conference on Computer Systems (EuroSys)*, 2010, pp. 41–54.

[54] K. Moore, J. Bobba, M. Moravan *et al.*, "LogTM: Log-based transactional memory," in *12th IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2006, pp. 254–265.

[55] T. Müller, A. Dewald, and F. Freiling, "AESSE: A cold-boot resistant implementation of AES," in *3rd European Workshop on System Security (EuroSec)*, 2010, pp. 42–47.

[56] T. Müller, F. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *20th USENIX Security Symposium*, 2011.

[57] National Vulnerability Database, "CVE-2014-0069," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0069.

[58] ——, "CVE-2014-0160," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160.

[59] ——, "CVE-2014-4653," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-4653.

[60] Y. Ni, A. Welc, A.-R. Adl-Tabatabai *et al.*, "Design and implementation of transactional constructs for C/C++," in *23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2008, pp. 195–212.

[61] T. Parker and S. Xu, "A method for safekeeping cryptographic keys from memory disclosure attacks," in *1st International Conference on Trusted Systems (INTRUST)*, 2010, pp. 39–59.

[62] E. Pattuk, M. Kantarcioglu, Z. Lin, and H. Ulusoy, "Preventing cryptographic key leakage in cloud virtual machines," in *23rd USENIX Security Symposium*, 2014.

[63] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *29th IEEE Symposium on Security*

and Privacy (S&P), 2008, pp. 233–247.

[64] N. Petroni and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *14th ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 103–115.

[65] T. Pettersson, "Cryptographic key recovery from Linux memory dumps," in *Chaos Communication Camp*, 2007.

[66] Primate Labs, "Geekbench 3," http://www.primatelabs.com/geekbench/.

[67] Roberto A. Vitillo, "Linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page.

[68] D. Samyde, S. Skorobogatov, R. Anderson, and J.-J. Quisquater, "On a new way to read data from memory," in *1st IEEE Security in Storage Workshop*, 2002, pp. 65–69.

[69] M. Seltzer, Y. Endo, C. Small, and K. Smith, "Dealing with disaster: Surviving misbehaved kernel extensions," in *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996, pp. 213–227.

[70] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2007, pp. 335–350.

[71] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[72] S. Sidiroglou and A. Keromytis, "Execution transactions for defending against software failures: Use and evaluation," *International Journal of Information Security*, vol. 5, no. 2, pp. 77–91, 2006.

[73] P. Simmons, "Security through Amnesia: A software-based solution to the cold boot attack on disk encryption," in *27th Annual Computer Security Applications Conference (ACSAC)*, 2011, pp. 73–82.

[74] sp00n, "Security Dynamics FTP server core problem," 1997, http://insecure.org/sploits/solaris.secdynamics.core.html.

[75] P. Stewin, "A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory," in *16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2013, pp. 1–20.

[76] P. Stewin and I. Bystrov, "Understanding DMA malware," in *9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2013, pp. 21–41.

[77] The MITRE Corporation, "CWE-212: Improper cross-boundary removal of sensitive data," 2013, https://cwe.mitre.org/data/definitions/212.html.

[78] ——, "CWE-226: Sensitive information uncleared before release," 2013, https://cwe.mitre.org/data/definitions/226.html.

[79] Unknown author, "Solaris (and others) ftpd core dump bug," 1996, http://insecure.org/sploits/ftpd.pasv.html.

[80] P. van Dijk, "Coredump hole in imapd and ipop3d in slackware 3.4," 1998, http://www.insecure.org/sploits/slackware.ipop.imap.core.html.

[81] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "PixelVault: Using GPUs for securing cryptographic operations," in *21st ACM Conference on Computer and Communications Security (CCS)*, 2014, pp. 1131–1142.

[82] A. Wang, M. Gaudet, P. Wu *et al.*, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 127–136.

[83] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *19th ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 157–168.

[84] R. Yoo, C. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel transactional synchronization extensions for high-performance computing," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[85] F. Zhang, "IOCheck: A framework to enhance the security of I/O devices at runtime," in *the International Workshop, in conjunction with 43rd IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–4.

[86] Y. Zhang, A. Juels, M. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *19th ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 305–316.

[87] F. Zyulkyarov, V. Gajinov, O. Unsal *et al.*, "Atomic Quake: Using transactional memory in an interactive multiplayer game server," in *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009, pp. 25–34.