This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2897666, IEEE Transactions on Dependable and Secure Computing

Mimosa: Protecting Private Keys against Memory Disclosure Attacks using Hardware Transactional Memory

Congwu Li, Le Guan, Jingqiang Lin, *Senior Member, IEEE,* Bo Luo, *Member, IEEE,* Quanwei Cai, Jiwu Jing, *Member, IEEE,* and Jing Wang

Abstract—Cryptography is essential for computer and network security. When cryptosystems are deployed in computing or communication systems, it is extremely critical to protect the cryptographic keys. In practice, keys are loaded into the memory as plaintext during cryptographic computations. Therefore, the keys are subject to *memory disclosure attacks* that read unauthorized data from RAM. Such attacks could be performed through software exploitations, such as OpenSSL Heartbleed, even when the integrity of the victim system's binaries is maintained. They could also be done through physical methods, such as cold-boot attacks, even if the system is free of software vulnerabilities. This paper presents *Mimosa*, to protect RSA private keys against both software-based and physical memory disclosure attacks. Mimosa uses hardware transactional memory (HTM) to ensure that (*a*) whenever a malicious thread other than Mimosa attempts to read the plaintext private key, the transaction aborts and all sensitive data are automatically cleared with hardware, 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 attacks. However, the fragility of TSX transactions introduces extra cache-clogging denial-of-service (DoS) threats, and attackers could sharply degrade the performance by concurrent memory-intensive tasks. To mitigate the DoS threats, we further partition an RSA private-key computation into multiple transactional parts by analyzing the distribution of aborts, while (sensitive) intermediate results are still protected across transactional parts. Through extensive experiments, we show that Mimosa effectively protects cryptographic keys against attacks that attempt to read sensitive data in memory, and introduces only a small performance overhead, even with concurrent cache-clogging workloads.

Index Terms—Cold-Boot Attack; CPU-Bound Encryption; DMA Attack; Memory Disclosure Attack; Transactional Memory.

1 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 the signing or decryption operations, the private keys are usually loaded into memory as plaintext, and become vulnerable to *memory disclosure attacks* that read unauthorized data in memory. Such attacks are launched through software exploitations. For instance, the OpenSSL Heartbleed vulnerability allows remote attackers to steal sensitive memory data [78]. Unprivileged processes exploit vulnerabilities [40, 63, 77, 79] to obtain unauthorized data in memory. The statistics on Linux show that, 16.2% of the vulnerabilities [24] can be exploited to read unauthorized data from the memory space of operating system (OS) kernels or user processes. Such attacks can be launched successfully, even if the integrity of the victim system's binaries is maintained at all times. So existing mechanisms such as buffer-overflow guards [22, 23, 105] and kernel integrity protections [46, 58, 84, 89], are ineffective against these "silent" attacks. Meanwhile, attackers with physical accesses are capable of bypassing all OS protections to directly read data from RAM chips, even if the system is free of the vulnerabilities mentioned above. Cold-boot attacks [41] "freeze" the RAM chips of a running computer, place them into another machine controlled by the attacker, and read the RAM contents.

This paper presents Mimosa that uses *hardware transactional memory* (HTM) to protect private keys against both software and physical memory disclosure attacks described above. We use Intel Transactional Synchronization eXtensions (TSX) [48], a commodity HTM solution in commercial-off-the-shelf (COTS) platforms. Transactional memory was originally proposed as a speculative memory access mechanism to boost the performance of multi-threaded applications [45]. An execution with transactional memory finishes successfully, only in the case of no data conflict; otherwise, all operations are discarded and the execution is rolled back. A data conflict happens, if multiple threads concurrently access the same memory location and at least one of them is a write operation. The strong *atomicity* guarantee provided by HTM is utilized to defeat illegal accesses to the memory space that contains sensitive data. Moreover, Intel TSX and most HTM are physically implemented in caches, so the computing is constrained entirely within CPUs, effectively

• *C. Li, L. Guan (co-first author), J. Lin (corresponding author), Q. Cai, J. Jing and J. Wang, are with Data Assurance and Communications Security Center, and also State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, China; Email: {cwli13, lguan, linjq, qwcai, jing, jwang}@is.ac.cn; and B. Luo is with Department of Electrical Engineering and Computer Science, University of Kansas, USA; Email: bluo@ku.edu.*

The preliminary version appeared under the title "Protecting Private Keys against Memory Disclosure Attacks using Hardware Transactional Memory" [38] in Proc. 36th IEEE Symposium on Security and Privacy, 2015 .

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2897666, IEEE Transactions on Dependable and Secure Computing

preventing cold-boot attacks on RAM chips.

We adopt the key-encryption-key structure in Mimosa – the RSA private keys in memory remain encrypted by an AES master key, when there is no signing or decryption request. Mimosa integrates TRESOR [76], a register-based AES cryptographic engine, to protect the AES key-encryption key always in debug registers that are only accessible with ring 0 privileges. The AES master key is derived from a password, input when the system boots. Meanwhile, if Mimosa is triggered for a request, the RSA private key is decrypted by the AES master key and then used as follows.

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 used to decrypt or sign messages. If the transaction is interrupted due to any reason (e.g., attack attempt, interrupt, or fault), a *hardware-enabled* abort handler clears all updated but uncommitted data in the transaction, which guarantees that the private key (and intermediate states) cannot be accessed by malicious processes. The abort processing is *non-maskable*, and triggered by HTM automatically. Before committing the computation result, all sensitive data are carefully cleared. So a software memory disclosure attack only obtains cleared data, even if it successfully reads from the memory addresses of the 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 RAM chips (see [51], Chapter 12.1.1). So Mimosa is immune to cold-boot attacks.

We implemented the Mimosa prototype with Intel TSX, but the design is applicable to other HTM implementations using on-chip caches [2, 54, 104] or store buffers [26, 42]. When the private-key computation is executed as an HTM transaction and the private key is decrypted (i.e., the data are updated) in the transactional execution, any attack attempt to access the private key results in data conflicts that abort the transaction. These HTM solutions are CPU-bound, so they are also effective against 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 transactional memory is originally proposed for speculatively running critical sections, a TSX transaction 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 intrinsics in OS functions, interrupts, preemption, and other unexpected aborts; otherwise, the transactional execution never commits. Moreover, when the Mimosa service is running concurrently with other memory-intensive processes, it has to deal with aborts due to the competition of cache resources, which hence introduces cache-clogging denial-of-service (DoS) threats. Compared with the preliminary design of Mimosa [38], we further partition a private-key computation into multiple transactional parts, to mask the fragility of TSX transactions and mitigate the impact from concurrent memory-intensive tasks, while (sensitive) intermediate results are encrypted across the transactional parts by the AES master key.

Mimosa is implemented as a Linux kernel module and exported as an OpenSSL-compatible cryptographic engine. We have evaluated the prototype on an Intel Core i7 4770S Haswell CPU. Through extensive validations, we confirm that no private key is disclosed under various memory disclosure attacks. Experiments show that Mimosa only introduces a small overhead to provide the security guarantees. Its performance is comparable to popular RSA implementations without additional protections, either in clean environments or with concurrent cache-clogging workloads.

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

The rest of the paper is organized as follows. Section 2 introduces the background and preliminaries. We present the Mimosa design and implementation details in Sections 3 and 4, respectively. The cache-clogging DoS threat is mitigated in Section 5. Experimental results are shown in Section 6, and the security analysis is in Section 7. We summarize related works in Section 8 and finally conclude the paper.

2 Background and Preliminaries

2.1 Memory Disclosure Attacks on Sensitive Data

These attacks are roughly classified into two categories: software-based and hardware (or physical) attacks.

Software Memory Disclosure Attack. Software vulnerabilities allows adversaries to read unauthorized data from the memory space of OS kernels or user processes, without modifying binaries. These vulnerabilities result from unverified inputs, isolation defects, memory dump, memory reuse or cross-use. OpenSSL Heartbleed attackers receive sensitive data by manipulating malformed TLS heartbeat requests [78]; or attackers exploit the kernel vulnerability [40] to read memory data. The un-initialization error and the ALSA bug [77, 79] lead to sensitive information leakage from kernel space. As a result of unintended software designs, such as core dump, hibernation and crash reports, the memory content could be swapped to disks [18], which may be accessible to attackers. Cryptographic keys are recovered from Linux memory dump files [86]. Some FTP and Email servers dump data to a directory accessible to adversaries [61, 95, 101], leaking passwords that are originally kept in memory. Finally, uncleared data buffers are subject to reuse or cross-use [72, 73], and the RSA private keys are disclosed from uncleared buffers through the Linux ext2 vulnerability [44].

Cold-Boot Attack. These attacks result from the remanence effect of semiconductor devices; that is, the contents of dynamic RAM (DRAM) chips gradually fade away. At low

1. Our solution reactively clears the memory to protect sensitive data whenever an attack attempt is conducted. 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.

temperatures the fading speed slows down significantly. So adversaries retrieve the remained data by cold-booting the running target computer and loading a malicious OS from an external storage [41]. Cold-boot attacks can be launched alternatively by placing the DRAM chips into another machine controlled by attackers. Such attacks require no account or credential on the target machine, and are launched even if the victim system is free of software vulnerabilities.

DMA Attack. Direct Memory Access (DMA) is designed to allow peripheral devices to bypass the OS and directly access memory for better performance. However, this feature is exploited in another type of physical memory attacks [97]. Read-only DMA attacks read out sensitive memory by DMA requests from Firewire or PCI interfaces [6, 11, 47], and the malicious behaviors do not need any "cooperation" of OSes. Advanced DMA attacks inject malicious binaries into the memory of victim computers by DMA requests, and then the injected codes access data in memory or registers [10].

2.2 CPU-Bound Solutions against Cold-Boot Attacks

While there are various solutions against software memory disclosure attacks [14, 28, 44, 83], the defense against cold-boot attacks is to bound the operations in CPUs. CPU-bound solutions avoid loading sensitive data into RAM chips, so cold-boot attacks fail. Register-based cryptographic engines [76, 93] implemented the AES algorithm entirely within CPUs. TRESOR stores an AES key in debug registers and Amnesia uses model-specific registers. Atomicity must be ensured during the encryption/decryption to avoid swapping the register states to memory.

PRIME [29], RegRSA [112] and Copker [37] extend the CPU-bound approach to RSA. The AES key protected by TRESOR is used as a key-encryption key to encrypt RSA private keys. In PRIME, the private key is decrypted into AVX registers and the RSA computations are performed in these registers. The performance is decreased to about 10% of traditional implementations, due to the limited size of registers. RegRSA improved PRIME by using more registers and encrypting sensitive intermediate states in memory, so the efficiency is enhanced. Copker employs CPU caches to perform the RSA computations against cold-boot attacks. It assumes the integrity of OS kernels without any memory disclosure vulnerabilities [40, 63, 77, 79], so Copker is not immune to software memory disclosure attacks.

2.3 Transactional Memory and Intel TSX

Transactional memory is a memory access mechanism, originally designed to improve the performance of concurrent threads and reduce programming efforts [45]. The 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.

Intel TSX provides *hardware-enabled* transactional memory [48]. Programmers specify critical sections for transactional executions, and the processor transparently performs data tracking, conflict detection, commit and roll-back. Data conflicts are detected on top of the cache-coherence protocol, at the granularity of cache lines. TSX keeps all updated but uncommitted data in L1D caches, and tracks a read-set (addresses that have been read from) and a write-set (addresses that have been written to) during the transaction. A conflict is detected if another core (*a*) reads from a memory location in the write-set, or (*b*) writes to a location in the write-set or read-set. In addition to data conflicts, other events abort a TSX transaction, including unfriendly instructions such as cache controls, operations on MMX states, and system activities such as interrupt and exception. There are also micro-architectural implementation dependent reasons.

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. The prefixes give hints to processors that the 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 called Restricted Transactional Memory (RTM), provides three new instructions (i.e., `XBEGIN`, `XEND` and `XABORT`) to start, commit, and abort a transactional execution. Programmers specify a *fallback function* as the operand of `XBEGIN`. The aborted execution jumps to the specified fallback function, so the programmers can implement customized codes to handle the exception.

3 System Design

This section presents the assumptions and security goals of Mimosa. We then introduce the system architecture, and some important design details.

3.1 Assumptions and Security Goals

Assumptions. We assume the correct hardware implementation of HTM (i.e., Intel TSX in the prototype system or others in the future). 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.

Attackers are assumed to be able to launch memory disclosure attacks. They can stealthily read memory data in OS kernels by exploiting memory disclosure vulnerabilities [40, 63, 77, 79], or launch cold-boot attacks [41]. They can eavesdrop the communication on the bus between the CPU and RAM chips. Mimosa attempts to defend against these "silent" memory attacks that read memory data without breaking the integrity of privileged binaries. We do not consider the multi-step attacks – the attackers first write malicious binaries into the victim system's kernel, and then access data via the injected codes. That is, we assume that the integrity of OS kernels is not compromised always, while memory disclosure vulnerabilities exist in the kernel. Kernel integrity can be guaranteed by existing mechanisms, such as TPM [100] during the initialization, and SecVisor [89], OSck [46], SBCFI [85], Lares [84] or kGuard [58] at runtime. Besides, the adversaries may perform any operations with non-root privileges, e.g., run concurrent memory-intensive tasks to compete for resources with Mimosa.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2897666, IEEE Transactions on Dependable and Secure Computing

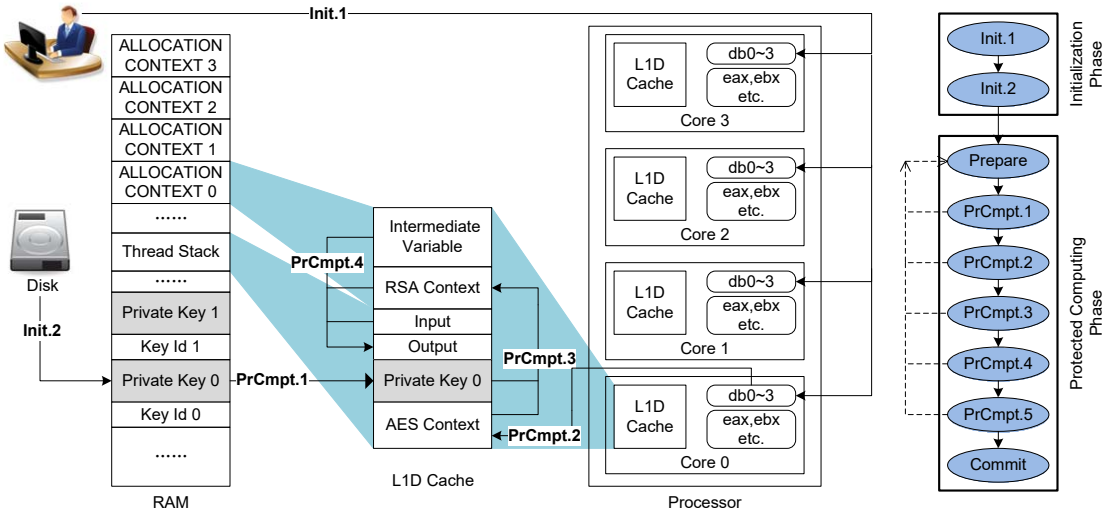


Fig. 1: Mimosa Overview

We employ TRESOR to protect the master key in privileged debug registers [76], so Mimosa inherits its assumptions. TRESOR and similar solutions assume no interface or vulnerability that allows attackers to access the debug registers. The access to these privileged registers is blocked by patching the `ptrace` system call (the only interface from user space), disabling loadable kernel modules and `kmem`, and removing JTAG ports (done in COTS products) [76, 93]. We do not impose additional assumptions on other registers, either privileged or not; that is, other registers are regularly accessed by a task within its context, subject to the control of CPU privilege rings.

Different from the existing mechanisms which attempt to detect or prevent software attacks (e.g., buffer-overflow guards [22, 23, 105]), Mimosa follows a different philosophy – it tries to "dance" with attacks. We do not prevent malicious read operations on protected sensitive data; however, even if an attacker exploits vulnerabilities to successfully circumvent the protections and read data from memory, Mimosa ensures that the attacker still cannot obtain the private keys that were stored at the accessed memory locations.

Security Goal. We design Mimosa with the following goals. (1) During each private-key operation, no thread other than the Mimosa 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. And (3) The sensitive data never appear on the RAM chips.

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

Mimosa does not specifically consider side-channel attacks on cryptographic engines. Such attacks will be countered by algorithm designs such as RSA blinding [15]. More discussions on side channels are included in Section 7.2.

3.2 The Mimosa Architecture

Mimosa adopts the common key-encryption-key structure. The AES master key is derived 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 the signing/decryption requests. When the Mimosa service is in idle, private keys remain encrypted by the master key.

The operations of Mimosa consist of two phases in Figure 1: an *initialization* phase and a *protected computing* phase. The initialization is executed only once when the system boots. It initializes the AES master key in debug registers. The protected computing phase is executed on each RSA private-key computation request, to perform the requested computations. All memory accesses during the protected computing phase are tracked to achieve our security goals.

Initialization Phase. It contains two steps. **Init.1** resembles TRESOR and executes completely in kernel space when the system boots. A command line prompt is set up for the user to enter a password. The AES master key is derived from the password, and copied to the debug registers of every CPU core. Then, all intermediate states of this derivation are carefully erased. The user is required to type in 4096 more characters to overwrite input buffers. We assume that there is no 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 RSA 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., a dedicated off-line machine.

Protected Computing Phase. When Mimosa receives a request from users, it uses the corresponding key to perform the private-key computation, and returns the result. Mimosa prepares the transactional execution, performs the computation, erases all sensitive data, and finally terminates the transaction to commit the result. It includes these 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 the 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 variables in caches and registers are erased, except the result.
- **Commit**: Finish the transaction and make the result available.

All memory accesses during this phase are strictly monitored by hardware. In particular, we declare a *transactional region* for this phase. During the transactional execution, all memory operations that might break Mimosa's security principles are detected by hardware: (*a*) any attempt to access the modified memory locations, i.e., the plaintext private key and intermediate states generated during the execution; and (*b*) cache eviction or replacement that synchronizes data in caches to RAM chips. Software memory disclosure attacks or DMA attacks on the plaintext private key and intermediate states, will cause such memory exceptions. In the case of no memory exception, the transaction commits and the result is returned to users; otherwise, the hardware-enabled abort handler is triggered automatically to clear all modified data. Then, it jumps to the program-specified fallback function (not shown in Figure 1); in the fallback function, we choose to retry immediately or take supplementary actions before retrying (see Section 4.3).

To take full advantage of multi-core processors, Mimosa is designed to support multiple computation tasks in parallel. A block of memory space is reserved for each core in the transactional region (i.e, the protected computing phase). The space is used mainly for the dynamic memory allocation in the RSA computations, and separated properly for each core to avoid unexpected data conflicts that lead to aborts (see Section 4.3 for details).

The Mimosa architecture is based on the general properties of HTM. Mimosa does not rely on any specific HTM implementation. It is expected that this architecture is applicable to any COTS HTM solution. In the remainder, we describe the implementation and evaluation of Mimosa with a commodity HTM product, i.e., Intel TSX.

4 Implementation

We first introduce the RTM interface and a naïve implementation of Mimosa as a Linux kernel module. We then examine the causes of the aborts that significantly reduces performance, and optimize the implementation to obtain the performance comparable to conventional RSA engines.

4.1 RTM Programming Interface

Mimosa utilizes Intel TSX as the underlying transactional memory primitives. We choose RTM as the HTM programming interface. With this flexible interface, we have controls over the fallback path.

RTM provides instructions (`XBEGIN`, `XEND` and `XABORT`) to start, commit and abort a TSX transaction. `XBEGIN` consists of a two-byte opcode `0xC7 0xF8` and an operand. The operand is a relative offset to the `EIP` register, to calculate the address of the *program-specified* fallback function. On aborts, the CPU immediately breaks the transaction and restores micro-architectural states. Then, the execution resumes at the fallback function. At the same time, the abort reason is marked in the corresponding bit(s) of the `EAX` register. The reason code in `EAX` is used for quick decisions at runtime; for example, the third bit indicates data conflicts, and the fourth indicates that the cache is full. However, this returned code does not precisely reflect every event [48]. For instance, the aborts due to unfriendly instructions or interrupts do not set any bit. In fact, Intel suggests performance monitoring for deep analyses when programming with TSX, before releasing the software.

We encapsulate the RTM instructions into C functions in Linux kernel. At the time of our implementation, we did not find any support for RTM in the main Linux kernel branch. Although Intel Compiler, Microsoft Visual Studio, and GCC have supports for RTM in user-space programming, they are not ready for kernel programming. We refer to the Intel manual [51] to implement the RTM intrinsics using inline assembler equivalents. The `_xbegin()` function to start a transaction is 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 until successfully commits. 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 the transaction and the return value is set properly (i.e. the abort status in `EAX`).

4.2 The Naïve Implementation

The AES master key is always protected in debug registers, and in the protected computing we adopted PolarSSL v1.2.10 as the base of our AES and RSA modules. PolarSSL is an efficient library with a small memory footprint. A smaller work-set means adequate cache resources to complete the transaction. In the long-integer module of PolarSSL, a piece of assembly codes uses the `MMX` registers. It is marked as unfriendly instructions of Intel TSX [48]. We replaced `MMX` with `XMM`. It needs only a little modification because both operands are supported in the SSE2 extension. The AES module of PolarSSL is an S-box-based implementation, but we improved it with AES-NI [49] to encrypt/decrypt data by the AES master key in debug registers.[2] It has three benefits. The memory footprint is reduced without

2. Newer versions of PolarSSL support AES-NI. We develop it independently to avoid shared memory.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2897666, IEEE Transactions on Dependable and Secure Computing

S-box, performance is boosted with hardware acceleration, and timing-based side channels of AES implementations [1, 7, 12] are eliminated by running in constant time.

We implement steps **PrCmpt.1** to **PrCmpt.5** as a function in C language: `mimosa_compute(int keyid, unsigned char *in, unsigned char *out)`. As mentioned above, **PrCmpt.5** erases all sensitive data before committing the transaction. The sensitive data appears in:

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

It seems straightforward to integrate the codes in the 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){
    if (_XBEGIN_STARTED == _xbegin())
        break;
}
mimosa_compute(keyid, in, out);
_xend();
```

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 demonstrate various causes that lead to aborts and our optimizations. We used the `perf` profiling tool [103] and Intel Software Development Emulator (SDE) v6.12 [99] for the purpose of abort reason discovering and performance tuning. The `perf` tool works with the Intel performance monitoring facility. It supports precise-event-based sampling (PEBS) that records the processor state once a particular event happens. We use the `RTM_RETIRED.ABORTED` event to capture aborts. This event occurs every time an RTM execution is aborted. Based on the dumped processor state, we are able to locate the abort reason and the eventing instruction pointer (IP) that causes the abort. SDE is the Intel official emulator for 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.

4.3 Performance Tuning

Avoiding Data Conflicts. We found that the modular exponentiation used the OS-provided memory allocation library which maintains shared meta data (e.g., the free list) for all threads. As a result, data conflicts happen when threads in different cores request for new memory simultaneously.

We let each Mimosa thread monopolize its own allocation context in the transactional region. We reserve a global array of allocation contexts, and each context is defined for one core. The first member in `ALLOCATION_CONTEXT` is aligned on a 64-byte boundary (i.e., a cache line), which is the granularity to track the read/write-sets. This prevents false data sharing between continuous contexts, which happens when two threads access their distinct memory locations in the same cache line.

When a Mimosa thread enters the transactional region and the memory allocation function is called, the thread first gets its core ID and uses it to locate its designated allocation context. It performs memory allocation in this context.

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

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

With this tuning, Mimosa works 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 any 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. SDE does not simulate interrupts. The private-key computation is time-consuming, so it is very likely that the transactional execution is interrupted by task scheduling on real hardware, which definitely causes aborts. Other interrupts also cause aborts. To give Mimosa enough time to complete computations, interrupts and preemption are temporarily disabled when it is in the transactional region. Existing CPU-bound cryptographic engines [29, 37, 76, 93, 112] disable interrupts to ensure *atomicity*, while Mimosa requires it for *efficiency* because Intel TSX ensures atomicity already.

Delay after Continuous Aborts. At this point, the *abort cycle ratio*, the number of CPU cycles in the aborted transactions divided by the total number of cycles in all transactions, is relatively high. The `perf` profiling tool is unable to provide obvious information on abort reasons. The eventing IPs recorded by PEBS spread across the transactional region. Most of reported reason codes are `ABORTED_MISC5`, which has a very ambiguous description by Intel.

- `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.

We suspect that the aborts were caused by non-maskable interrupts (NMIs), but it is ruled out after examining the NMI counter through the `/proc/interrupts` interface. As stated before, Intel provides no guarantee as to whether

3. Mimosa is modified slightly, conforming to SDE in user space.

a TSX transaction will successfully commit and there are lots of implementation-specific reasons that cause aborts [48].[4]

We notice that Intel recommends a delay before retrying if the abort is caused by data conflicts [51]. 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 [109] and also verified in our experiments. After extensive experiments balancing the throughput under single-threaded and multi-threaded scenarios, we identify 10 clock ticks as an optimized value for this delay.[5] We would like to emphasize that the 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 does not succeed if we skip any of the previous steps.

After these tunings, most of the remaining aborts occur at the very beginning of the transactions. 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 very low, as more than 95% of the cycles are used in each 2048-bit RSA decryption/signing. The simulation showed that our implementation is correct, but it may be impossible to completely avoid all aborts: (*a*) the Intel official tools are unable to identify or provide the details of aborts; and (*b*) Intel TSX and the speculation 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 cause aborts.

4.4 Utility Issues

We need a trusted machine (e.g., a dedicated off-line PC) to generate private keys, and a to-be-protected system running the Mimosa service. The preparation utility in the off-line machine generates RSA key pairs and encrypts them by an AES key derived from the same password. The encrypted key file is then copied to the to-be-protected machine.

Mimosa is implemented as a module patched to Linux kernel v3.13.1. It supports 1024/2048/3072/4096-bit RSA, and provides services to user space through the `ioctl` system call. Based on the request code and key ID, Mimosa outputs the public key, or performs a private-key operation and outputs the result. The `ioctl` interface is further encapsulated into an OpenSSL engine, facilitating the integration of Mimosa into applications in an OpenSSL-compatible way. We use this API to integrate it into Apache HTTPS servers.

5 Cache-Clogging DoS Attacks and Partitioned Protected Computing

We investigate the aborts in the presence of cache-clogging DoS attacks (or concurrent memory-intensive tasks), and partition the RSA private-key operation into multiple transactional parts to mitigate the impact of such threats.

4. We have contacted Intel by Email. Until the submission of this manuscript, we did not receive any reply.

5. For the HTM feature in zEC12 systems, IBM also suggests a random delay before retrying a transaction on aborts [54]; and the optimal delay depends on abort reasons, CPU designs and configurations.

5.1 Aborts with Concurrent Memory-Intensive Tasks

The Mimosa implementation in Section 4 provides efficiency comparable to conventional RSA engines, but the fragility of TSX transactions introduces extra cache-clogging DoS threats. When it is running concurrently with memory-intensive tasks, the abort cycle ratio increases and the performance is sharply degraded, because the read/write-set is tracked within CPU caches which are limited in size and shared among all cores. When we launched the STREAM memory suite of Geekbench 3 concurrently with the 2048-bit RSA decryption service (4 threads) of Mimosa, more than 50% of CPU cycles are wasted in aborted transactional executions, while it is less than 5% in clean environments. Intel CPUs implement 8-way set associative L1D caches, 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 are attributed to transactional executions. So a process with intensive memory accesses may halt the Mimosa service, because there is a great chance that this process evicts the cache lines that Mimosa is occupying due to the shared caches.

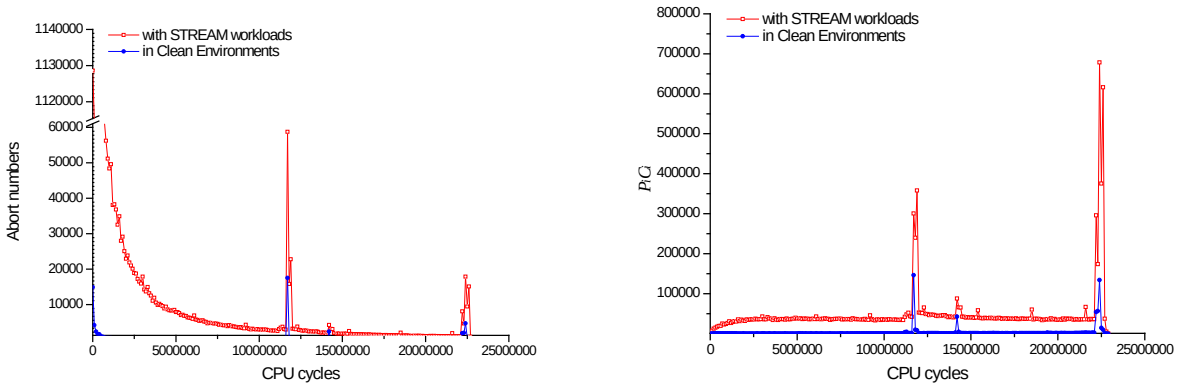
To improve the performance concurrently with memory-intensive tasks and mitigate the cache-clogging DoS threat, we firstly reduce the memory footprint of RSA private-key operations: (*a*) an element of an RSA private key is decrypted only it is to be used, and (*b*) several variables are stored at the same memory location after some of them are no longer used. But the performance improvement is rather negligible: the abort cycle ratio of 2048-bit RSA decryptions (4 threads) is reduced from 57.34% to 56.52% with STREAM workloads, and from 4.56% to 3.12% in clean environments.

We further partition a heavy RSA private-key computation into multiple transactional parts: since aborts are inevitable in transactional executions, especially in the presence of concurrent memory-intensive tasks, we preserve the expensive intermediate results across transactional parts. This partition is designed with the following intentions: (*a*) even if the whole RSA private-key computation does not finish, some expensive intermediate results are preserved for future retrying; and (*b*) less memory and fewer CPU cycles are needed in each transactional part, so that it is more possible to commit the transactions.

We try to find the optimal position(s) to partition the protected computation, by analyzing the distribution of aborts. Figure 2 shows the average numbers of aborts at different positions i, and the position is measured from the beginning of a private-key computation, in 10^5 CPU cycles. We recorded aborts for 800,000 successful transactional executions of 2048-bit RSA decryptions in clean environments and with STREAM workloads, respectively, and A_i denotes the average number of aborts at i for one successful execution. There are several peak values of A_i in Figure 2, because the memory amount of the read/write-set increases non-linearly as a CRT-enabled RSA private-key operation [59] sequentially conducts two modular exponentiations and one modular multiplications. When it is starting a modular exponentiation (or multiplication), the memory requirement increases sharply; but the occupied memory keeps almost unchanged within one exponentiation/multiplication.

On average it takes $C_i = \frac{i+i\sum_{j\geq i} A_j + \sum_{j<i} jA_j}{1+\sum_{j\geq i} A_j}$ CPU

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2897666, IEEE Transactions on Dependable and Secure Computing



(a) Abort distribution of one successful transaction

(b) Wasted CPU cycle at different positions

Fig. 2: Abort Distribution and P_iC_i of the Unpartitioned Mimosa

cycles to successfully execute to i, because the $1 + \sum_{j \geq i} A_j$ executions across i result from (*a*) one successful transaction, which takes i cycles to execute across i, (*b*) $\sum_{j \geq i} A_j$ unsuccessful ones that have executed across i but abort at j ($j \geq i$), each of which also takes i cycles, and (*c*) $\sum_{j<i} A_j$ unsuccessful ones that abort at j ($j < i$). The probability that an execution aborts at i, is $P_i = \frac{A_i}{1+\sum_{j \geq i} A_j}$, for it executes across i for $1 + \sum_{j \geq i} A_j$ times and aborts at i for A_i times.

The variation of P_iC_i is also shown in Figure 2, and it describes the CPU cycles wasted on average due to the aborts at i. We shall partition the transaction exactly before any peak value of P_iC_i, to preserve these expensive intermediate results that would otherwise be discarded. Two significant peak values of P_iC_i appear exactly after the two modular exponentiations of CRT-enabled RSA private-key operations (see Algorithm 1), or when it is starting the second modular exponentiation and the final modular multiplications. The two peak values appear at the same positions in these two extreme scenarios, either in clean environments or with STREAM workloads. We also launched the `mbw` memory benchmark and the memory test (4 threads, total 8G bytes with 256-byte blocks, random read/write) of SysBench concurrently with Mimosa, respectively, and find the same positions of the significant peak values of P_iC_i. Moreover, these two significant peaks after the modular exponentiations are confirmed in our experiments of 3072/4096-bit RSA.

Thus, we analyze two partitioned Mimosa implementations: one partitions the RSA private-key operation into only two transactional parts after the first modular exponentiation, and the other includes three parts partitioned after each of the two modular exponentiations. Figure 3 shows P_iC_i of two partitioned implementations, where the dotted lines are the positions to partition an RSA private-key computation. The peak values of P_iC_i are significantly reduced after the partition, either in clean environments or with STREAM workloads: they are about one or two orders of magnitude less than those of the unpartitioned version in Figure 2. When it runs concurrently with memory-intensive tasks, the abort cycle ratio of the partitioned version is only 14% for two transactional parts and 11% for three parts, respectively, compared with 57% for the unpartitioned version.

We do not partition the RSA private-key operation into more transactional parts, because the peak values in P_iC_i are not so significant after the partition with three parts; for instance, the peak values in Figure 3(d) are only about two times of the average value. Meanwhile, each partition brings a small overhead (summarized in Section 5.2), which may sometimes counteract the benefits of the partition design. The comprehensive performance evaluation in Section 6 shows that the partitioned implementation with three transactional parts outperforms the two-part version in some scenarios, but performs even a little worse in others.

ALGORITHM 1: RSA Decryption Partitioned into Three Parts

Input: $ciphertext$, $encpdp$
Output: $t1cipher$
$(p, dp) = AESDecrypt(encpdp)$
$t1 = ciphertext^{dp} \mod p$
$t1cipher = AESEncrypt(t1)$

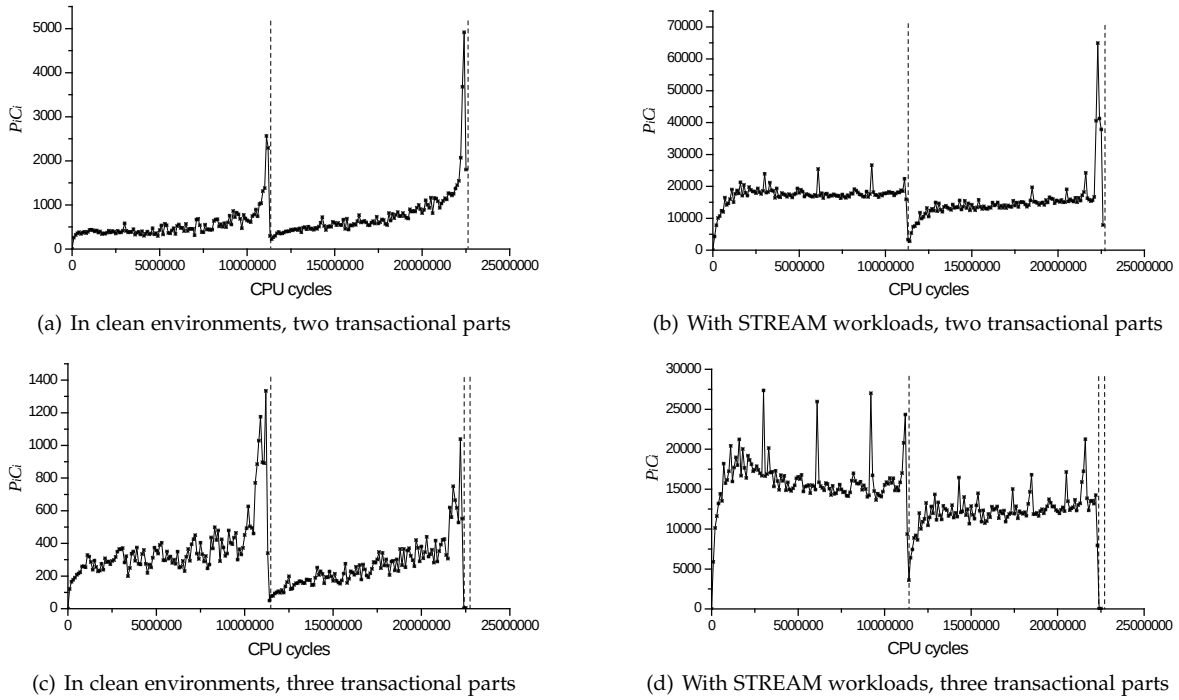
Input: $ciphertext$, $encqdq$
Output: $t2cipher$
$(q, dq) = AESDecrypt(encqdq)$
$t2 = ciphertext^{dq} \mod q$
$t2cipher = AESEncrypt(t2)$

Input: $ciphertext$, $t1cipher$, $t2cipher$, $encpqq$
Output: $plaintext$
$(p, q, qinv) = AESDecrypt(encpqq)$
$(t1, t2) = AESDecrypt(t1cipher, t2cipher)$
$plaintext = (t1 - t2) * qinv \mod p$
$plaintext = t2 + plaintext * q$

5.2 Partitioned Transactional Execution with Protected Intermediate Results

To protect (sensitive) intermediate results across transactional parts, which might be exploited to disclose the private key, the intermediate results are encrypted by the AES master key before it commits in each part except the last one, then decrypted and used in the next. In each transactional part, all updated data except the (intermediate) results are erased carefully before committing the transaction (see Section 4.2 for details). Algorithm 1 describes the steps of partitioned CRT-enabled RSA private-key operations, with three transitional parts. The CRT-enabled RSA private-key octuple [59] is denoted as $(n, e, d, p, q, dp, dq, qinv)$, while

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2897666, IEEE Transactions on Dependable and Secure Computing



(a) In clean environments, two transactional parts

(b) With STREAM workloads, two transactional parts

(c) In clean environments, three transactional parts

(d) With STREAM workloads, three transactional parts

Fig. 3: P_iC_i of Partitioned Mimosa Implementations

$encpdp$, $encqdp$ and $encpqq$ are the encrypted copies of private elements (p, dp), (q, dq) and $(p, q, qinv)$, respectively. The first and second transactional parts perform two modular exponentiations, and other computations are finished in the third. The partition with two parts has a similar structure and is not presented due to the page limit.

Each CPU core is configured with its own memory allocation context (see Section 3.2). The encrypted intermediate results are stored in this designated space, so all transactional parts of one private-key computation are restricted on the same core by setting the Mimosa thread's CPU affinity.

Figure 4 shows the final code snippet of Mimosa partitioned with three transactional parts. The additional overheads of the partition include: (*a*) AES encryption/decryption of the intermediate results across transactional parts, (*b*) AES decryption of some private-key elements for more than one times, and (*c*) start and commit more transactional executions, which involve expensive instructions. The next section shows that, the partitioned versions of Mimosa produces almost the same performance as the unpartitioned implementation. More importantly, the partition approach works much better on the competition of cache resources, effectively mitigating the cache-clogging DoS threat due to the fragility of TSX transactions. The partition also improves the performance when it is integrated in user-space applications. When integrated in applications, the Mimosa service runs with concurrent tasks and then the partition design works better; moreover, the partitioned RSA private-key operation is interruptable among transactional parts, so that the application functions are served more timely.

6 Performance Evaluation

This section presents the experiment results on the performance of Mimosa. Experiments are performed on a machine with an Intel Core i7 4770S CPU (4 cores, 3.4 GHz), running a patched Linux kernel v3.13.1. Both the unpartitioned and partitioned implementations are evaluated: Mimosa executes each RSA private-key operation in one transaction, Mimosa_Partitioned_2 adopts the partition design with two transactional parts and Mimosa_Partitioned_3 with three parts. We compared them with: (*a*) PolarSSL version 1.2.10 with default configurations, (*b*) Mimosa_No_TSX, the same as Mimosa but not in transactional executions, by turning off the `TSX_ENABLE` switch in Figure 4 (i.e., PolarSSL in the kernel, disabling interrupts and preemption), and (*c*) Copker [37]. We used 2048-bit RSA keys in the these experiments, except the scalability evaluation in Section 6.4.

6.1 Local Performance

Mimosa (and other approaches) ran as local RSA decryption services, called by an evaluator in user space. We measured the number of decryption operations per second, at different concurrency levels. In Figure 5, all approaches exhibit similar performance except Copker, which uses only one core due to the L3 cache shared by all cores of Intel Core i7 4770S CPUs: the Copker core works in the `write-back` cache-filling mode, and forces 3 other cores into the `no-fill` mode [37]. Only one Copker task works and other tasks have to wait, so the performance is about 1/4 of PolarSSL.[6] Mimosa, Mimosa_Partitioned and Mimosa_No_TSX performs even better than PolarSSL, for PolarSSL is subject to more task scheduling while preemption is disabled in all versions of Mimosa. Mimosa, Mimosa_Partitioned_2 and Mimosa_Partitioned_3 offer almost the same performance

6. On the Intel Core2 Q8200 CPU, 4 cores share two separate L1/L2 cache-sharing sets and no L3 cache, and the performance of Copker is about 1/2 of PolarSSL [37].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2897666, IEEE Transactions on Dependable and Secure Computing


```
mimosa_RSA(keyid) {
#ifdef PARTITION // Switch of Mimosa_Partitioned
    set_task_cpu_core(smp_processor_id());
    // Set the thread's CPU affinity
    transaction_execute(RSA_decrypt_part1, keyid);
    transaction_execute(RSA_decrypt_part2, keyid);
    transaction_execute(RSA_decrypt_part3, keyid);
    clear_task_cpu_core();
#else
    transaction_execute(RSA_decrypt, keyid);
#endif
}

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

Fig. 4: The Partitioned Implementation of Mimosa

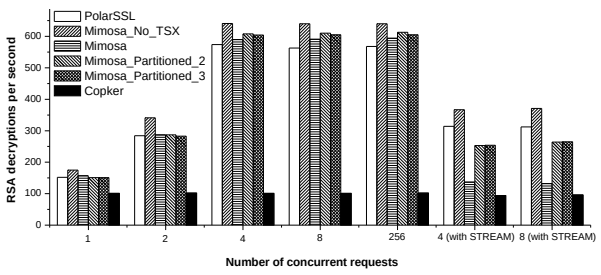


Fig. 5: Local Performance of RSA Decryption

in the clean environment. The additional overheads by the partition are negligible, compared with the expensive RSA computations. Mimosa_Partitioned_2 performs a little better than Mimosa_Partitioned_3, because more operations due to the partition are needed for three parts, but the benefits are not so imperative in this scenario.

We then evaluated how seriously a memory-intensive program would impact (different versions of) Mimosa, by launching the STREAM memory test concurrently with Mimosa. In this experiment, 4 kinds of STREAM workloads were performed on all CPU cores, resulting in about 10 GB memory data transferred per second. The clean machine supports a maximum transfer rate of 13.7 GB/s. All approaches suffer a significant performance decrease except Copker, where about three fourth of the computation resources are reserved. The average performance of Mimosa falls to 137 decryptions per second in Figure 5. That is a degradation of 77.0%, compared with the original result of 596 per second. The number of Mimosa_Partitioned_2 decreases from 613 to 253, and the degradation is only 58.8%, while the number of Mimosa_Partitioned_3 decreases by 58.2%, from 605 to 253. Meanwhile, the performance of Mimosa_No_TSX decreases by 42.0%; and the degradation of PolarSSL is 44.8%. So about 45% of the degradation is caused by the resource occupation of STREAM, and only about 30% is caused by the transaction aborts in Mimosa due to intensive memory access, while this impact is mitigated to less than 15% by the partition design. We have tried other different memory-intensive programs, and all of them have performance impact less than STREAM.

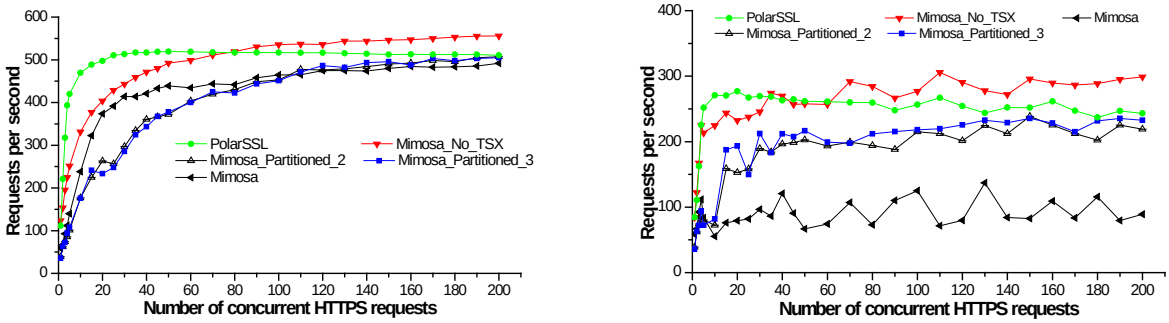
We measured the abort cycle ratios of the approaches with TSX. When running in the clean environment, under all concurrency levels, the abort cycle ratio of Mimosa is always under 5%, and the number of any partitioned version is less than 3%. The abort cycle ratio of Mimosa (4 threads) raises to 57% in the case of concurrent memory-intensive tasks, compared with only 14% for Mimosa_Partitioned_2 and only 11% for Mimosa_Partitioned_3. The partition of TSX transactions offers effective resilience against the DoS threat exploiting the limited cache resources, while the performance keeps almost the same in cases of no such threat.

6.2 HTTPS Throughput and Latency

In this experiment, each evaluated approach served as the RSA decryption module in the Apache HTTPS server, and then we measured the throughput and latency. The web page in this experiment was 4K bytes in size. The server and the client were located in an isolated Gigabit Ethernet.

The client ran ApacheBench sending requests at different concurrence levels, and the numbers of HTTPS requests handled per second are shown in Figure 6(a). The maximum throughput of Mimosa loses 17.6% of its local capacity, while it is 17.2% for Mimosa_Partitioned_2 and Mimosa_Partitioned_3 loses 16.5%. The numbers of Mimosa_No_TSX and PolarSSL are 13.5% and 6.5%, respectively. From the results, we estimate that the first 6.5% loss for all approaches should be attributed to the unavoidable overhead of HTTP, TLS and network packet transmission. Disabling preemption has a negative impact on concurrent tasks, so all versions of Mimosa, either partitioned or not, become worse than user-space PolarSSL; but Mimosa_No_TSX performs still a little better than PolarSSL after the number of concurrent requests reaches 80. The additional 11% loss of capacity in Mimosa shall be caused by aborted CPU cycles and it is improved a little in the partitioned versions, because when integrated in HTTPS servers, the Mimosa service is actually running concurrently with more tasks, so the partitioned versions outperform the unpartitioned one. On the other hand, when the concurrency level is low, the CPU cores are not utilized fully,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2897666, IEEE Transactions on Dependable and Secure Computing



(a) In clean environments (b) With STREAM workloads

Fig. 6: HTTPS Throughput

because multiple transactional parts of one RSA private-key computation must be switched on the same core, but the threads of Apache HTTPS servers are not uniformly allocated on all cores. So the advantage appears only after the number of concurrent requests reaches 100 and there are always available tasks on each core. Note that, if we could allocate the HTTPS workloads uniformly as we did in the local performance evaluation in Section 6.1, the improvement of the partition design shall appear when the concurrency level is low. Figure 6(b) shows the HTTPS throughput with STREAM workloads. We compare the average results of two scenarios, which are calculated when the throughput is stable (i.e., the number of concurrent requests varies from 100 to 200). The results of PolarSSL, Mimosa_No_TSX, Mimosa, Mimosa_Partitioned_2, and Mimosa_Partitioned_3 decrease by 48.8%, 76.2%, 47.1%, 44.6%, and 46.6%, respectively. About 45% of the decrease for all approaches is caused by the STREAM workloads, and the additional 30% for Mimosa results from the fragility of TSX transactions.

We used curl (one client, the keep-alive option disabled) to measure HTTPS latency. The average TLS handshake times were 9.98ms, 9.04ms, 10.94ms, 10.48ms and 10.43ms, when PolarSSL, Mimosa_No_TSX, Mimosa, Mimosa_Partitioned_2 and Mimosa_Partitioned_3 served in the HTTPS server, respectively. Different versions of Mimosa perform a little worse than PolarSSL, and the partition improves the results. We use ApacheBench to stress the HTTPS server to measure its 95th percentile latency. As shown in Figure 7(a), the negative impact of disabling preemption and aborted cycles in different versions of Mimosa is acceptable. The 95th percentile latency of Mimosa is about 1.6 times of PolarSSL, and it is improved to 1.4 times by the partition design after the number of concurrent requests reaches 100. The performance improvements by the partition, can be explained as above in the HTTPS throughput experiment. The results of HTTPS latency with STREAM workloads are shown in Figure 7(b). The 95th percentile latency of Mimosa increases sharply: it becomes averagely 3.2 times of PolarSSL, while the results of Mimosa_No_TSX, Mimosa_Partitioned_2, and Mimosa_Partitioned_3 are only 1.2, 1.8, and 1.6 times, respectively, when the number of concurrent requests varies from 100 to 200. Compared with the results in Figure 7(a), the 95th percentile latency of Mimosa with STREAM workloads becomes 4.6 times averagely, while others are 2.1 - 2.8 times only.

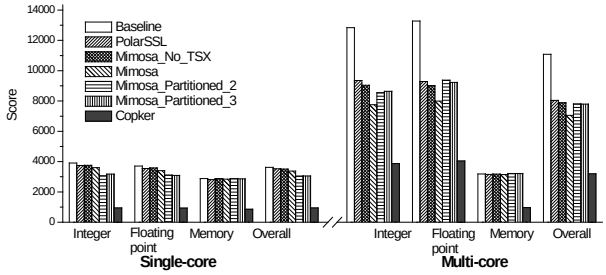


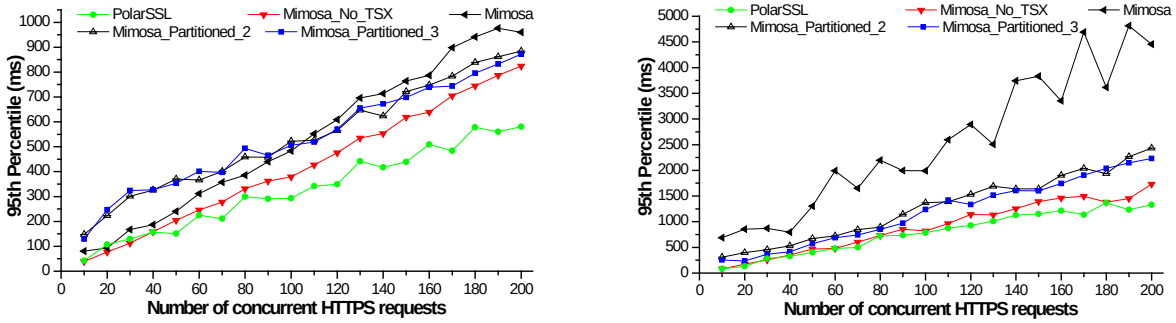
Fig. 8: Geekbench 3 Scores under RSA Computations

6.3 Impact on Concurrent Processes

The Geekbench 3 benchmark is used to evaluate how (different versions of) Mimosa influenced concurrent applications. Running concurrently with the RSA private-key computations of each evaluated solution, Geekbench 3 measures the machine's integer, floating point and memory performance, i.e., the capacity remained for concurrent processes. The Geekbench 3 scores for both the single-core mode and the multi-core mode are shown in Figure 8. The baseline score was measured without any process except Geekbench 3, indicating the machine's full capacity. Each approach was measured when the benchmark was running concurrently with Apache at the workload of 80 HTTPS requests per second. All evaluated approaches should work at the same computation workload. Since the maximum throughput of Copker is around 100 HTTPS requests per second, we pick 80 requests per second in this experiment.

The scores in Figure 8, represent the performance of integer instruction, floating-point instruction and memory bandwidth. Overall, the scores of PolarSSL, Mimosa, Mimosa_Partitioned_2/3, and Mimosa_No_TSX are very close, either in the single-core mode or the multi-core mode, except Copker. The scores of partitioned Mimosa are always a little better than those of unpartitioned version. 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, which does not exist in the single-core mode. User-space PolarSSL introduces a little less impact on concurrent processes than kernel-space approaches (i.e., Mimosa, Mimosa_Partitioned_2/3, and Mimosa_No_TSX) where preemption is disabled. In Figure 5, we find that preemption-disabled approaches are a little more efficient because more resources are occupied

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2897666, IEEE Transactions on Dependable and Secure Computing



(a) In clean environments (b) With STREAM workloads

Fig. 7: HTTPS Latency

TABLE 1: Local Performance (decryptions/sec) for Different RSA Key Lengths

Key Length (bits)	1024	2048	3072	4096
Mimosa_No_TSX	3798	640	212	95
Mimosa	3726	594	153	30
Mimosa_Partitioned_2	3742	613	199	75
Mimosa_Partitioned_3	3732	605	195	68

by them. However, it also means that concurrent processes cannot be served in time, as shown in Figure 8. Last, we find a significant drop in Copker for both the single-core mode and the multi-core mode, because other CPU cores are forced to enter the `no-fill` mode when Copker is running. That is, Geekbench 3 runs without caches at times.

6.4 Scalability

We evaluated the performance of Mimosa with growing RSA key lengths, to analyze its potential capacity to other algorithms with more memory and heavier computation.

In this experiment, Mimosa ran locally to accomplish the maximum speed, called by the 256-threaded evaluator in user space. In Table 1, the improvement by the partition (or the gap between the unpartitioned implementation and the partition versions) become more and more remarkably, as the key length grows. When more memory is accessed, the TSX transactions become more fragile and the partition design is more useful. The performance of Mimosa_Partitioned_2 (the best version of Mimosa in this scenario) decreases at a similar pace with Mimosa_No_TSX, as the key length grows from 1024 bits to 2048, 3072 and 4096 bits – the performance ratios of Mimosa_Partitioned_2 to Mimosa_No_TSX are 99%, 96%, 94% and 80%, respectively. Thus, the fragility of TSX transactions has not become the bottleneck to support stronger keys for up to 4096-bit RSA, especially with the partition design. Anyway, as more memory is required, more CPU cycles are wasted in aborted transactional executions.

We measured the size of dynamically allocated memory in a transaction which approximates the whole work set for 4096-bit RSA computations. The allocated memory size was about 9.3K bytes, far less than the supported write-set size of Intel TSX evaluated in [69], 26K bytes. So there is still a great potential for supporting other memory-hungry algorithms.

7 Security Analysis and Discussion

This section validates that Mimosa achieves the security goals in Section 3.1. Then, the remaining attack surfaces are discussed, and we compare Mimosa with existing defenses against cold-boot attacks (and also other attacks) on RSA private keys. We also discuss the applicability of Mimosa.

7.1 Validation and Analysis

To validate that software memory disclosure attacks cannot obtain the sensitive data of Mimosa, we implemented a privileged "attack" program (the validator), which actively reads the memory addresses used in Mimosa through the `/dev/mem` interface. The memory locations are fixed once Mimosa has been loaded. `/dev/mem` is a virtual device that provides an interface to access the system's memory. 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 user-space threads repeatedly calling the Mimosa services at the full speed. Throughout the experiment, the validator returned cleared data only. Note that this "attack" program is more powerful than real-world memory disclosure attacks, because it runs with root privileges and knows the exact address of sensitive data. As a comparison, when we disabled the TSX protection, almost every access obtained the plaintext RSA private keys.

We 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 kernel memory and the register states of other cores when the system crashes. It sends *non-maskable* interprocessor interrupts (IPIs) to other cores to halt the system.[7] We crashed the system by writing 'c' to `/proc/sysrq-trigger`, while Mimosa was running intensively. We searched for the RSA private key and the AES master key in the dump file. The AES key has two forms: the original 128-bit key and 10-round key schedule. First, for the AES key schedule, we used AESKeyFinder [41] to analyze the captured image, and no matching key schedule was found. For the original AES key and the RSA private key, the image had no matching of any

7. NMIs destroy atomicity for only local interrupt is disabled. This is the reason that CPU-bound solutions suggest modifying NMI handlers to immediately clear all sensitive keys [37, 93]. However, in Mimosa, the private key and intermediate states are automatically cleared once NMIs happen, eliminating the need to modify NMI handlers.

keys (including p, q, d and other private CRT elements). We never found a bit sequence that overlaps for more than 3 bytes with any key. On the contrary, when this experiment was conducted on Mimosa_No_TSX, we got many copies of the AES key and the RSA private key. They came from three sources. First, `Kdump` dumped the register states when the system crashed. Second, interrupted threads left the decrypted keys uncleared. Third, the control blocks of threads contained the register states as a result of context switching.

Read-only DMA attacks [97] might attempt to access the modified memory data (i.e., private keys) in Mimosa. DMA requests on such memory addresses synchronize data from caches to RAM chips, and this synchronization aborts the transactional execution and clear the private keys [48].

We did not launch a cold-boot attack or probe the bus to validate that these are no data leaked to RAM chips. Recent studies show that, the memory contents are scrambled on DDR3 RAM chips, so cold-boot attacks cannot directly read the plaintext data [35] but at most 128 bytes of known plaintext are required in a descrambling attack to recover the memory content [4]. The Mimosa prototype works with two 4G-byte DDR3 RAM chips, and we did not re-build the special descrambling tool to recover the private keys in RAM chips. Meanwhile, according to the Intel manual (see [51], Chapter 12.1.1), when cache eviction in the write-set happens, a transaction aborts immediately, and modified data are discarded inside L1D caches. The plaintext private keys and intermediate states are in the write-set, for they are generated after the transaction starts. So these data appear nowhere other than L1D caches. Note that, it is a necessary requirement to correctly implement HTM. If a modified cache line is evicted outside the boundary of Intel TSX, its value will be available to other components – an obvious contradiction to the nature of transactional memory.

During the transactional execution of Mimosa, the plaintext private key is kept in caches but not RAM chips [36, 48]. Attackers might reboot or power off the computer, attempting to dump the cache content in a way similar to the cold-boot attacks on RAM chips. Such attacks do not work, since internal caches are invalid after power-up or reset and there is no interface to access data in caches from outside [52].

7.2 Remaining Attack Surface

Attackers might exploit side channels to compromise the keys. Cache-based side channels [7, 12] do not exist in Mimosa, because AES-NI is free of such attacks [49] and the RSA computations are performed entirely in caches. Other side channels of timing [3, 15], electromagnetic fields [30], ground electric potential [31], power [81] or acoustic emanations [32], can be prevented by RSA blinding [15]. The random bits in RSA blinding will be also protected by the AES master key [39], against memory disclosure attacks.

Mimosa assumes the integrity of OS kernels, so integrity protections (e.g., SecVisor [89], SBCFI [85], OSck [46], Lares [84], and kGuard [58]) shall work complementarily. While the kernel integrity solutions protect the Mimosa binaries from being modified, Mimosa defeats memory disclosure attacks not violating the integrity of binaries. [10] exhibits an advanced DMA attack that injects malicious codes into an OS kernel (i.e., breaks the integrity) and then accesses the AES key in debug registers. Fortunately, the DMA attacks are countered by various solutions [66, 96, 97, 110].

The vulnerabilities of Meltdown [67] and Spectre [60] enable illegal or speculative read operations and leak the results through cache side channels [68, 108]. The read operations of Meltdown run with the privileges of the attacker process, while the operations of Spectre do with the victim's privileges. In the future, we will investigate whether such read operations abort the TSX transaction after or before the cache state is changed (i.e., the read result is leaked or not).

7.3 Comparison with Software Cryptographic Engines against Cold-Boot Attacks

There are RSA implementations on common OSes against cold-boot attacks, namely, PRIME [29], RegRSA [112], Copker [37] and the proposed work. These solutions adopt the same key-encryption-key structure – an AES master key is kept in privileged registers throughout the operation of the system, and the RSA private key is decrypted on demand to perform requested operations. Table 2 summaries four approaches in terms of OS assumption, efficiency and RSA implementation. Hardware assumptions are not shown in the table, such as Intel TSX, cache-filling modes, CPU privilege rings, etc. With the hardware support from Intel TSX, Mimosa significantly outperforms other solutions. Moreover, because the private-key computation is implemented in C language, it is much easier for Mimosa and Copker to support other algorithms such as DSA and ECDSA.

TRESOR is used in all solutions to protect the AES master key, the security of which depends on the integrity of the kernel executable without any interfaces to privileged debug registers. TRESOR is resilient to memory disclosure vulnerabilities in OS kernels. However, the security of RSA private-key operations is very different:

PRIME and **RegRSA** uses unprivileged registers to store RSA private keys, and requires the atomicity guarantee of private-key computations; otherwise, the registers may be accessed by attackers that interrupt the computations. Due to the limited size of registers, it is extremely difficult or even impossible to support longer RSA private keys.

Copker is vulnerable to software-based memory disclosure attacks. It depends on the kernel isolation (i.e., the assumption of no memory disclosure vulnerabilities in the kernel), because illegal memory read operations by other cores to access the memory address of private keys will synchronize the plaintext private key in caches to RAM chips, when the Copker core is decrypting or signing messages. Then, in such cases the sensitive data are subject to cold-boot attacks.

Mimosa assumes kernel integrity, and its atomicity is guaranteed by hardware. It is resilient to memory disclosure vulnerabilities in OS kernels, even if the memory space is used in the private-key computations. Mimosa also requires the atomicity guarantee by OS for efficiency.

7.4 Comparison with SGX

Intel Software Guard eXtensions (SGX) builds hardware-enabled user-space containers, isolated from other processes and OS kernels [53]. Confidentiality and integrity of the SGX enclaves are maintained against privileged malware, so it is

TABLE 2: Comparison of the RSA Implementations against Cold-boot Attacks

Solution	OS Assumption		Performance	RSA Algorithm
	Master Key	Private Key	Compared with PolarSSL	Implementation
Mimosa	X+D	X$^{\diamond}$	Comparable	C Language
PRIME	X+D	X+A	Approximately 1/9†	Assembly Code
RegRSA	X+D	X+A	Approximately 3/4‡	Assembly Code
Copker	X+D	X+A+R	Approximately 1/4*	C Language

X: Integrity of executable binaries in OS kernels. **D**: No illegal access to debug registers.
A: Atomicity guarantee of private-key computations. **R**: No illegal memory read operation.
$\diamond$: Mimosa requires the atomicity guarantee by OSes for efficiency, not for security.
† ‡: The numbers are drawn directly from [29] and [112], respectively.
$*$: The number of separate cache sets divided by that of cores [37]. An Intel Core i7 CPU has 4 cores with shared L3 caches.

also utilized for cryptographic engines [56, 65, 87]. While Mimosa ensures only the confidentiality of private keys against memory disclosure attacks, both confidentiality and integrity of data and binaries in SGX enclaves are provided. The performance penalty in log systems by SGX is about 5% [56], and our experiment that implements 2018-bit RSA computations in SGX enclaves as a local service, shows that the overhead by SGX is less than 1%.

SGX cannot work in kernel space, and it protects a *user-space* process mainly against the underlying (malicious or curious) OSes and hypervisors. So SGX is more suitable for the applications on public clouds. On the contrary, Mimosa implements computations in *kernel space* and provides services to user-space processes, so it is more applicable to scenarios where the user-space processes and the OS belong to the same owner, or to implement services in trusted OS kernels. Besides, although both TSX and SGX are hardware features of Intel CPUs, SGX additional requires the special BIOS support to configure processor reserved memory (PRM) for SGX enclaves. Currently, the number of SGX-compatible mainboards is still limited.

7.5 Applicability

Although the Mimosa prototype is implemented with Intel TSX using the RTM interface,[8] our design is applicable to other platforms. In particular, if the protected computing is executed as a transaction using the HLE interface of Intel TSX, `XTEST` will be used to determine whether it is in a transactional execution or not. If it is in a normal execution (i.e., the transaction aborts), the protected computing will not continue and the transactional execution will be retried.

Most HTM solutions share a similar programming interface. In other HTM implementations, the counterparts of `XBEGIN` and `XEND` are easily identified, and the abort processing conforms to the Mimosa design. In the HTM facility of IBM zEC12 [54], transactions are defined by `TBEGIN` and `TEND`. On aborts, the `PC` register is restored to the instruction immediately after `TBEGIN`, and a condition code is set to a non-zero value. A program tests the condition code after `TBEGIN` to start the transactional execution if `CC=0` or branch to the fallback function if not. AMD proposed its HTM extension, called Advanced Synchronization Facility (ASF), but currently products are not ready. ASF provides similar instructions to start and commit a transactional execution (i.e., `SPECULATE` and `COMMIT`) and tracks memory accesses in caches [2]. ASF has a slightly different feature that all to-be-tracked memory must be explicitly specified.

Finally, most HTM implementations use on-chip components for the transaction execution [2, 26, 42, 54, 104], so they can also work with Mimosa to prevent cold-boot attacks.

8. In August 2014, Intel announced a bug in the TSX implementation, and suggested disabling TSX on the affected CPUs via a microcode update [50]. In our experiments, the Mimosa prototype works well. TSX is still supported in newer Intel CPUs, e.g., Core M-5Y71 in Q4 2014, Core M7-6Y75 in Q3 2015, and Core i7-6785R in Q2 2016.

8 Related Work

8.1 Attack and Defense on Sensitive Memory Data

More copies of cryptographic keys in memory result in higher leakage risk [44]. To reduce the risk, secure deallocation [19] erases data either on deallocation or within a predictable period, to reduce the number of copies of sensitive data in unallocated memory. Harrison et al. keep only one copy of cryptographic keys in allocated memory [44]. 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 [83]. Scrash [14] removes sensitive data from crash reports in the case of program failures. Mimosa follows the same spirit to control sensitive data in memory, and we employ HTM to enforce this principle.

TRESOR [76] and Amnesia [93] store AES keys only in registers, against the cold-boot attacks on full-disk encryption. FrozenCache [82] stores AES keys in caches and configures the cache-filling modes to prevent the keys from being flushed to RAM chips. The CPU-bound approach is extended to the RSA algorithm. Using the AES key protected by TRESOR as a key-encryption key, PRIME [29] implemented the RSA computation in AVX registers while Copker [37] did it in caches. RegRSA [112] improved PRIME by using more registers and encrypting intermediate results in memory, so the efficiency is improved. Mimosa implements RSA against cold-boot attacks, but provides much better performance. The register-based engines also prevent read-only DMA attacks [6, 11, 97] that passively read data from memory. Advanced attacks [10] actively inject malicious codes into OS kernels by DMA requests, and then access registers. These DMA attacks are detected or restricted by monitoring bus activities [96], leveraging SMM [110], configuring IOMMU [97], or the timed challenge-response protocol [66].

PixelVault uses GPUs as the container for cryptographic computing [102] – during the initialization all sensitive data

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2897666, IEEE Transactions on Dependable and Secure Computing

and executable binaries are loaded into the caches and registers of GPUs, so (malicious) binaries on CPUs cannot access these data and binaries on GPUs. GPUs are dedicated for cryptographic computing in PixelVault, while Mimosa dynamically builds secure containers within CPUs. Sentry [21] employs the cache-locking feature of ARM CPUs to defeat cold-boot and DMA attacks: sensitive data are locked in caches while the encrypted copies are on RAM chips. PhiRSA [113] exploits the vector instructions of Intel Xeon Phi to implement high-performance RSA computations.

There are SGX-based security solutions [5, 88, 94], and [56, 65, 87] implements cryptographic engines in SGX enclaves. There are vulnerabilities found in SGX enclaves [13, 64, 92, 106, 107], leaking sensitive data. Flicker [71] built dynamical isolated execution environments, utilizing the hardware feature of late launch and attestation in Intel Trusted eXecution Technology (TXT) and AMD Secure Virtual Machine (SVM). The overhead to initialize and exit an SGX enclave or a Flicker piece is heavier than a TSX transaction [48, 53, 71], so Mimosa is more suitable for frequently-called kernel modules. ARM TrustZone provides hardware isolation between execution domains. TrustOTP builds a TrustZone-based computing environment against malicious OSes [98]. CaSE [111] extends TrustOTP by constraining computations in caches, against both cold-boot attacks and software memory attacks. Memory resources occupied by the TrustZone secure domain cannot be used by other domain, while the execution environment of Mimosa is dynamically built on-demand and released in idle. These solutions show the same tendency of building security systems on top of hardware features. Last, RamCrypt [34] and HyperCrypt [33] are software-based memory encryption for Linux processes against software and physical memory disclosure attacks; but the performance penalty is significant.

8.2 Transactional Memory Application and Exploitation

Transactional memory boosts thread-level parallelism, and is applied in database systems [57] and game servers [70, 114]. Transactional memory improves the multi-threaded support in dynamic binary translation to ensure the correct executions of concurrent threads [20].

By maintaining shared resources in the read/write-set, TMI enforces authorization policies once such a resource is accessed [8, 9]. TMI and Mimosa depend on transactional memory to inspect the access to sensitive resources. TMI enforces authorization policies on every access, while Mimosa ensures confidentiality by clearing sensitive keys once any illegal read operation occurs. TSX-CFI [75] maps control flow transitions into TSX transactions, and violations of the intended control flow graph will trigger aborts. TxIntro [69] 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. It monitors the read-set to detect concurrent update operations that cause inconsistence, while Mimosa monitors the write-set to detect illegal read operations.

Utilizing the property that the Intel TSX abort handling is triggered by hardware and bypasses the underlying OS, T-SGX [91] isolates an SGX enclave against the page-fault side channel attacks by malicious OSes, and Deja Vu [17] builds a trustworthy timer to detect such attacks. Cloak [36] presents a framework against various cache-based side channels, by utilizing TSX to perform cryptographic computations in caches. On the contrary, exploiting the timing differences in TSX abort handlers for different memory faults, a side channel is constructed to break kernel address space layout randomization [55]. TSX tracks the read-set in L3 caches which are shared among all cores, so an abort due to cache eviction on the read-set provides information about other task's cache accesses to construct side channels [27].

8.3 Transactional Memory Implementation

Transactional memory designs are proposed, from hardware solutions [26, 42, 48, 54, 104] to software-based solutions [16, 43, 80, 90] and hybrid schemes [25, 62, 74]. HTM usually updates data temporarily in CPU-bound caches or store buffers before the transaction commits, and discards the updated data on aborts. LogTM updates memory directly and saves unmodified values in a per-thread log [74]; on aborts, the state is restored by inspecting through the logs.

9 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. 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, a commodity HTM solution. To mitigate the cache-clogging DoS threats due to the fragility of TSX transactions, we further partition an RSA private-key computation into multiple transactional parts by analyzing the distribution of aborts, while (sensitive) intermediate results are protected across these parts. We have simulated the 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. The performance evaluation shows that Mimosa exhibits comparable efficiency with conventional RSA implementations.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China (Award 61772518), and National 973 program of China (Award 2013CB338001).

*Bibliography

[1] O. Aciicmez, W. Schindler, and C. Koc, "Cache based remote timing attack on the AES," in *CT-RSA*, 2007.
[2] AMD, "Advanced synchronization facility, proposed architectural specification (revision 2.1)," 2009.

[3] C. Arnaud and P.-A. Fouque, "Timing attack against protected RSA-CRT implementation used in PolarSSL," in *CT-RSA*, 2013.

[4] J. Bauer, M. Gruhn, and F. Freiling, "Lest we forget: Cold-boot attacks on scrambled DDR3 memory," *Digital Investigation*, vol. 16, pp. S65–S74, 2016.

[5] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," in *USENIX OSDI*, 2014.

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

[7] D. Bernstein, "Cache-timing attacks on AES," 2004.

[8] A. Birgisson and U. Erlingsson, "An implementation and semantics for transactional memory introspection in Haskell," in *ACM PLAS*, 2009.

[9] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode, "Enforcing authorization policies using transactional memory introspection," in *ACM CCS*, 2008.

[10] E.-O. Blass and W. Robertson, "TRESOR-HUNT: Attacking CPU-bound encryption," in *ACSAC*, 2012.

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

[12] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *CHES*, 2006.

[13] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *USENIX WOOT*, 2017.

[14] P. Broadwell, M. Harren, and N. Sastry, "Scrash: A system for generating secure crash information," in *USENIX Security*, 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 *et al.*, "The Atomos transactional programming language," in *ACM PLDI*, 2006.

[17] S. Chen, X. Zhang, M. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with Deja Vu," in *ACM AsiaCCS*, 2017.

[18] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *USENIX Security*, 2004.

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

[20] J.-W. Chung, M. Dalton, H. Kannan, and C. Kozyrakis, "Thread-safe dynamic binary translation using transactional memory," in *IEEE HPCA*, 2008.

[21] P. Colp, J. Zhang *et al.*, "Protecting data on smartphones and tablets from memory attacks," in *ASPLOS*, 2015.

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

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

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

[25] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *ASPLOS*, 2006.

[26] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski, "Early experience with a commercial hardware transactional memory implementation," in *ASPLOS*, 2009.

[27] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX," in *USENIX Security*, 2017.

[28] A. Dunn, M. Lee *et al.*, "Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels," in *USENIX OSDI*, 2012.

[29] B. Garmany and T. Müller, "PRIME: Private RSA infrastructure for memory-less encryption," in *ACSAC*, 2013.

[30] D. Genkin, L. Pachmanov, and I. Pipman, "Stealing keys from PCs by radio: Cheap electromagnetic attacks on windowed exponentiation," in *CHES*, 2015.

[31] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs," in *CHES*, 2014.

[32] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *Crypto*, 2014.

[33] J. Götzfried, N. Dörr, R. Palutke, and T. Müller, "HyperCrypt: Hypervisor-based encryption of kernel and user space," in *ARES*, 2016.

[34] J. Götzfried, T. Müller, G. Drescher, S. Nürnberger, and M. Backes, "RamCrypt: Kernel-based address space encryption for user-mode processes," in *ACM AsiaCCS*, 2016.

[35] M. Gruhn and T. Müller, "On the practicability of cold boot attacks," in *ARES*, 2013.

[36] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *USENIX Security*, 2017.

[37] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with private keys without RAM," in *NDSS*, 2014.

[38] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang, "Protecting private keys against memory disclosure attacks using hardware transactional memory," in *IEEE S&P*, 2015.

[39] L. Guan, J. Lin, Z. Ma, B. Luo, L. Xia, and J. Jing, "Copker: A cryptographic engine against cold-boot attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 742–754, 2016.

[40] 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.

[41] J. Halderman, S. Schoen *et al.*, "Lest we remember: Cold boot attacks on encryption keys," in *USENIX Security*, 2008.

[42] L. Hammond, V. Wong *et al.*, "Transactional memory coherence and consistency," in *ISCA*, 2004.
[43] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *ACM PPoPP*, 2005.
[44] K. Harrison and S. Xu, "Protecting cryptographic keys from memory disclosure attacks," in *IEEE/IFIP DSN*, 2007.
[45] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993.
[46] O. Hofmann, A. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with OSck," in *ASPLOS*, 2011.
[47] D. Hulton, "Cardbus bus-mastering: 0wning the laptop," in *Annual ShmooCon Convention*, 2006.
[48] Intel, "Chapter 8: Intel transactional memory synchronization extensions," in *Intel architecture instruction set extensions programming reference*, 2012.
[49] ——, "Intel advanced encryption standard (AES) new instructions set," 2012.
[50] ——, "Desktop 4th generation Intel Core processor family," 2014.
[51] ——, "Intel 64 and IA-32 architectures optimization reference manual," 2014.
[52] ——, "Intel 64 and IA-32 architectures software developer's manual," 2014.
[53] ——, "Intel software guard extensions programming reference," 2014.
[54] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for IBM System Z," in *IEEE/ACM MICRO*, 2012.
[55] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with Intel TSX," in *ACM CCS*, 2016.
[56] V. Karande, E. Bauman, Z. Lin, and L. Khan, "SGX-Log: Securing system logs with SGX," in *ACM AsiaCCS*, 2017.
[57] T. Karnagel, R. Dementiev *et al.*, "Improving in-memory database index performance with Intel transactional synchronization extensions," in *IEEE HPCA*, 2014.
[58] V. Kemerlis, G. Portokalidis, and A. Keromyti, "kGuard: Lightweight kernel protection against return-to-user attacks," in *USENIX Security*, 2012.
[59] C. Koc, "High-speed RSA implementation," RSA Laboratories, Tech. Rep., 1994.
[60] P. Kocher, D. Genkin *et al.*, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, 2018.
[61] V. Kolontsov, "WU-FTPD core dump vulnerability (the old patch doesn't work)," 1997, http://insecure.org/sploits/ftp.coredump2.html.
[62] S. Kumar, M. Chu, C. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *ACM PPoPP*, 2006.
[63] M. Lafon and R. Francoise, "CAN-2005-0400: Information leak in the Linux kernel ext2 implementation," 2005, http://www.securiteam.com.
[64] J. Lee, J. Jang *et al.*, "Hacking in darkness: Return-oriented programming against secure enclaves," in *USENIX Security*, 2017.
[65] H. Li, J. Lin, B. Li, and W. Cheng, "PoS: Constructing practical and efficient public key cryptosystems based on symmetric cryptography with SGX," in *ICICS*, 2018.
[66] Y. Li, J. McCune, and A. Perrig, "VIPER: Verifying the integrity of peripherals' firmware," in *ACM CCS*, 2011.
[67] M. Lipp, M. Schwarz *et al.*, "Meltdown," *ArXiv e-prints*, 2018.
[68] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. Lee, "Last-level cache side-channel attacks are practical," in *IEEE S&P*, 2015.
[69] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen, "Concurrent and consistent virtual machine introspection with hardware transactional memory," in *IEEE HPCA*, 2014.
[70] D. Lupei, B. Simion *et al.*, "Transactional memory support for scalable and transparent parallelization of multiplayer games," in *EuroSys*, 2010.
[71] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4, 2008, pp. 315–328.
[72] MITRE, "CWE-212: Improper cross-boundary removal of sensitive data," 2013, https://cwe.mitre.org/data/definitions/212.html.
[73] ——, "CWE-226: Sensitive information uncleared before release," 2013, https://cwe.mitre.org/data/definitions/226.html.
[74] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, "LogTM: Log-based transactional memory," in *IEEE HPCA*, 2006.
[75] M. Muench, F. Pagani, Y. Shoshitaishvili, C. Kruegel, G. Vigna, and D. Balzarotti, "Taming transactions: Towards hardware-assisted control flow integrity using transactional memory," in *RAID*, 2016.
[76] T. Müller, F. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *USENIX Security*, 2011.
[77] National Vulnerability Database, "CVE-2014-0069," 2014, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0069.
[78] ——, "CVE-2014-0160," 2014, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160.
[79] ——, "CVE-2014-4653," 2014, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-4653.
[80] Y. Ni, A. Welc *et al.*, "Design and implementation of transactional constructs for C/C++," in *ACM OOPSLA*, 2008.
[81] Y. Oren and A. Shamir, "How not to protect PCs from power analysis," in *Crypto - Rump Session*, 2006.
[82] J. Pabel, "Frozencache: Mitigating cold-boot attacks for full-disk-encryption software," in *Chaos Communication Congress*, 2010.
[83] T. Parker and S. Xu, "A method for safekeeping cryptographic keys from memory disclosure attacks," in *INTRUST*, 2010.
[84] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *IEEE S&P*, 2008.

[85] N. Petroni and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *ACM CCS*, 2007.
[86] T. Pettersson, "Cryptographic key recovery from Linux memory dumps," in *Chaos Communication Camp*, 2007.
[87] L. Richter, J. Götzfried, and T. Müller, "Isolating operating system components with Intel SGX," in *SysTEX*, 2016.
[88] F. Schuster, M. Costa *et al.*, "VC3: Trustworthy data analytics in the cloud using SGX," in *IEEE S&P*, 2015.
[89] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *ACM SOSP*, 2007.
[90] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.
[91] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs," in *NDSS*, 2017.
[92] S. Shinde, Z.-L. Chua, V. Narayanan, and P. Saxena, "Preventing your faults from telling your secrets," in *ACM AsiaCCS*, 2016.
[93] P. Simmons, "Security through Amnesia: A software-based solution to the cold boot attack on disk encryption," in *ACSAC*, 2011.
[94] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *ACM CCS*, 2015.
[95] sp00n, "Security dynamics FTP server core problem," 1997, http://insecure.org/sploits/solaris.secdynamics.core.html.
[96] P. Stewin, "A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory," in *RAID*, 2013.
[97] P. Stewin and I. Bystrov, "Understanding DMA malware," in *DIMVA*, 2013.
[98] H. Sun, K. Sun, Y. Wang, and J. Jing, "TrustOTP: Transforming smartphones into secure one-time password tokens," in *ACM CCS*, 2015.
[99] A. Tal, "Intel software development emulator," 2012, http://software.intel.com/en-us/articles/intel-software-development-emulator.
[100] Trusted Computing Group, "Trusted platform module 2.0: A brief introduction," 2016.
[101] P. van Dijk, "Coredump hole in imapd and ipop3d in Slackware 3.4," 1998, http://www.insecure.org/sploits/slackware.ipop.imap.core.html.
[102] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "PixelVault: Using GPUs for securing cryptographic operations," in *ACM CCS*, 2014.
[103] R. Vitillo, "Linux profiling with performance counters," 2014, https://perf.wiki.kernel.org/.
[104] A. Wang, M. Gaudet *et al.*, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *PACT*, 2012.
[105] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *ACM CCS*, 2012.
[106] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves," in *ESORICS*, 2016.
[107] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE S&P*, 2015.
[108] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.
[109] R. Yoo, C. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel transactional synchronization extensions for high-performance computing," in *SC*, 2013.
[110] F. Zhang, "IOCheck: A framework to enhance the security of I/O devices at runtime," in *International Workshop, in conjunction with IEEE/IFIP DSN*, 2013.
[111] N. Zhang, K. Sun, W. Lou, and Y. Hou, "CaSE: Cache-assisted secure execution on ARM processors," in *IEEE S&P*, 2016.
[112] Y. Zhao, J. Lin, W. Pan, C. Xue, F. Zheng, and Z. Ma, "RegRSA: Using registers as buffers to resist memory disclosure attacks," in *IFIP SEC*, 2016.
[113] Y. Zhao, W. Pan, J. Lin, P. Liu, C. Xue, and F. Zheng, "PhiRSA: Exploiting the computing power of vector instructions on Intel Xeon Phi for RSA," in *SAC*, 2016.
[114] F. Zyulkyarov, V. Gajinov *et al.*, "Atomic Quake: Using transactional memory in an interactive multiplayer game server," in *ACM PPoPP*, 2009.

Congwu Li is a PhD student at Institute of Information Engineering, Chinese Academy of Sciences. He is interested in software security.

Le Guan is an assistant professor at Department of Computer Science, the University of Georgia. He received her PhD degree from Institute of Information Engineering, Chinese Academy of Sciences in 2015.

Jingqiang Lin received his MS and PhD degrees from Graduate University of Chinese Academy of Sciences in 2004 and 2009, respectively. He is a full professor at Institute of Information Engineering, Chinese Academy of Sciences. His research interest includes system security and applied cryptography.

Bo Luo is an associate professor at Department of Electrical Engineering and Computer Science, the University of Kansas. He received his PhD degree from Pennsylvania State University in 2008. His research interest includes security and data science.

Quanwei Cai is an assistant professor at Institute of Information Engineering, Chinese Academy of Sciences. He received his PhD degree from Institute of Information Engineering in 2015. His research interest is network security.

Jiwu Jing received his MS and PhD degrees from Graduate University of Chinese Academy of Sciences in 1990 and 2003, respectively. He is a full professor at Institute of Information Engineering, Chinese Academy of Sciences. He is interested in network and system security.

Jing Wang received her PhD degree from Institute of Information Engineering, Chinese Academy of Sciences in 2015.