Copker: Computing with Private Keys without RAM

Le Guan[†,‡,♮], Jingqiang Lin[†,‡,§], Bo Luo[♯], Jiwu Jing[†,‡]
† Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, CHINA
‡ State Key Laboratory of Information Security, Institute of Information Engineering,
Chinese Academy of Sciences, CHINA
♮ University of Chinese Academy of Sciences, CHINA
♯ Department of Electrical Engineering and Computer Science, The University of Kansas, USA
Email: lguan@is.ac.cn, linjq@is.ac.cn, bluo@ku.edu, jing@is.ac.cn
§ Corresponding author

***Abstract*—Cryptographic systems are essential for computer and communication security, for instance, RSA is used in PGP Email clients and AES is employed in full disk encryption. In practice, the cryptographic keys are loaded and stored in RAM as plain-text, and therefore vulnerable to physical memory attacks (e.g., cold-boot attacks). To tackle this problem, we propose Copker, which implements asymmetric cryptosystems entirely within the CPU, without storing plain-text private keys in the RAM. In its active mode, Copker stores kilobytes of sensitive data, including the private key and the intermediate states, only in on-chip CPU caches (and registers). Decryption/signing operations are performed without storing sensitive information in system memory. In the suspend mode, Copker stores symmetrically encrypted private keys in memory, while employs existing solutions to keep the key-encryption key securely in CPU registers. Hence, Copker releases the system resources in the suspend mode. In this paper, we implement Copker with the most common asymmetric cryptosystem, RSA, with the support of multiple private keys. We show that Copker provides decryption/signing services that are secure against physical memory attacks. Meanwhile, with intensive experiments, we demonstrate that our implementation of Copker is secure and requires reasonable overhead.**

Keywords*—*Cache-as-RAM; cold-boot attack; key management; asymmetric cryptography implementation.

I. Introduction

In computer and communication systems, cryptographic protocols are indispensable in protecting data in motion as well as data at rest. In particular, asymmetric cryptography is the foundation of a number of Internet applications. For instance, secure Email systems (PGP [20] and S/MIME [42]) are used to exchange encrypted messages and verify the identities of the senders. Meanwhile, SSL/TLS [16, 19] is widely adopted in secure HTTP [43], e-commerce, anonymous communications [17], voice over IP (VoIP) [13] and other communication systems. The security of such protocols relies on the semantic security of asymmetric cryptographic algorithms and the confidentiality of private keys. In practice, when the cryptographic modules are loaded, the private keys are usually stored in the main random-access-memory (RAM) of a computer system. Although various mechanisms have been proposed for memory protection, unfortunately, the RAM is still vulnerable to physical attacks. For instance, when the adversaries have physical access to a running computer, they can launch cold-boot attacks [23] to retrieve the contents of the main memory. Such attacks completely bypass memory protection mechanisms at operating system (OS) level. Therefore, any content, including cryptographic keys, stored in the memory could be extracted even though the adversaries do not have any system privilege in the target machine. The compromised private keys could be further exploited to decrypt messages eavesdropped from network communications, or to impersonate the owners of the private keys.

Access control, process isolation and other memory protection mechanisms at OS level cannot prevent cold-boot attacks, since the attackers usually reboot the machine with removable disks, or load the physical memory modules to their own machines to get a dump of the memory content. On the other hand, approaches based on memory management (e.g., the one-copy policy [24]) mitigate the problem by increasing the difficulty to find the private keys. Such methods are moderately effective for partial memory disclosure. Unfortunately, a successful cold-boot attack generates a dump of the entire physical memory, so that all "hidden" information are disclosed. More recently, TRESOR and Amnesia [38, 45] propose to store symmetric keys (e.g., AES keys) and perform encryption/decryption operations completely in CPU registers, so that keys are not loaded into the main memory during the process. The solutions are effective in protecting symmetric keys (typically not longer than 256 bits) against cold-boot attacks. However, they are not suitable for asymmetric cryptography, since private keys are too long to fit into CPU registers: RSA Laboratories [30] and NIST [3] recommend a minimum key length of 2048 bits for RSA private keys. Meanwhile, a 2048-bit RSA private key block needs at least 1152 bytes to work with Chinese remainder theorem (CRT)[1], and the intermediate states (in decryption or signing) need at least 512 bytes of additional storage (see Section II-A for details).

In this paper, we present a mechanism named *Copker* to perform asymmetric cryptographic algorithms without using RAM, and hence defeat against cold-boot attacks. In particular,

NDSS '14, 23-26 February 2014, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2014.23125

[1] CRT makes the computation approximately four times faster than that does not use CRT [32].

we implement RSA, the most prevalent asymmetric cryptographic algorithm, on multi-core CPU systems running Linux OS. During computation, Copker stores keys and intermediate states in on-chip CPU caches and registers, and computes with private keys entirely on the CPU. Therefore, plain-text private keys are never loaded to the RAM. To achieve this goal, Copker designs the following mechanisms: (1) during decryption/signing, the stack is switched so that all variables are directed into a reserved address space within caches; (2) the Copker task enters an atomic section, so that it cannot be suspended and the sensitive variables are never swapped to RAM; (3) other cores that share on-chip caches with the core running Copker are set to the no-fill mode during the computation, so that any task on these cores would not trigger cache replacement; and (4) private keys are either dynamically loaded into caches or encrypted in RAM, hence, the cache is occupied only when necessary.

We also designed a real-time checker program to verify that no plain-text sensitive data has been leaked to RAM during the stress test. The design goal of Copker is to defend against physical attacks on RAM. Hence, we assume a trustworthy OS kernel, that is, all binaries and processes with root privileges are trusted. The prototype system is implemented as a customized Linux kernel, but it can be ported to a trustworthy OS such as seL4 [31]. Moreover, since the asymmetric cryptographic algorithm in Copker is written in C language, hence, it is easier to be extended (compared with assembly language in [38, 45]) to support more cryptographic algorithms.

Our contributions are three-fold: (1) We are the first to propose an architecture to support asymmetric decryption/signing without using RAM. We keep private keys and intermediate variables/states in CPU caches and registers, so that confidential information is never loaded to RAM. (2) We implement the designed architecture, and demonstrate its security through security analysis as well as experimental validation. (3) Through intensive experiments, we show that our secure asymmetric decryption/signing scheme requires reasonable overhead. The rest of the paper is organized as follows: Section II presents the background about asymmetric cryptographic algorithms and caches. The design and implementation of Copker are described in Section III and IV, respectively. Section V evaluates Copker in term of validity and performance, followed by the security analysis in Section VI. Finally, Section VII surveys the related work and Section VIII draws the conclusion.

II. Background

A. RSA

RSA is the most prevalent asymmetric cryptographic algorithm for both encryption/decryption and signing/verification [44]. A typical RSA private key block is an octuple $(n, e, d, p, q, dp, dq, qinv)$, where (n, e) denotes the public key, d denotes the private key, and other variables are private parameters enabling the CRT speed-up. The key-length of the RSA key, denoted as L, is the length of n, which is 2048 bits or 3072 bits. The length of d is also L, while p, q, dp, dq and $qinv$ are all $L/2$ in length. The length of e is usually negligible (e.g., 64 bits). Therefore, a 2048-bit RSA private key needs at least $4.5L$, i.e., 1152 bytes of storage.

Algorithm 1: RSA Decryption with CRT

Input: $ciphertext$, n, e, d, p, q, dp, dq, $qinv$
Output: $plaintext$

1 $t1 \leftarrow String2Integer(ciphertext)$
2 $t2 \leftarrow t1^{dp} \ mod \ p$
3 $t3 \leftarrow t1^{dq} \ mod \ q$
4 $t1 \leftarrow (t2 - t3) * qinv \ mod \ p$
5 $t1 \leftarrow t3 + t1 * q$
6 $plaintext \leftarrow Integer2String(t1)$

To implement the RSA algorithm in computer systems, more memory in addition to the private key block is required to store temporary variables. The pseudo-code in Algorithm 1 shows the RSA decryption process with the CRT speed-up. From the pseudo-code, we can find that at least 3 intermediate variables are needed: $t1$ is L in length, while $t2$ and $t3$ are $L/2$. Therefore, for a 2048-bit RSA decryption, at least $6.5L = 1664$ bytes are needed to store the private key block and intermediate variables. Moreover, the pseudo code in Algorithm 1 only shows the major steps in the decryption process. When we consider the detailed implementation in each step, such as modular exponentiations and multiplications, more memory will be needed.

Moreover, commercial and well-designed RSA implementations may require even more spaces to support additional features. For example, Montgomery reduction [36], which can be used to accelerate modular multiplications, needs 3 long integers to store Montgomery values. To defend against timing attacks, RSA blinding is usually enabled [12], which requires extra memory space as well.

B. CPU Cache

On-chip cache is introduced to make up the speed gap between CPU and RAM. At present, the frequency of CPU is much higher than that of the memory bus. For example, Intel Core i7-2600 CPU has 4 cores running at 3.4GHz, while the bus frequency of a high-end DDR3-1600 RAM is 800MHz only. The discrepancy is primarily caused by the inherent physical limitations of the dynamic RAM (DRAM) hardware. On the other hand, the cache is a small amount of high-speed static RAM (SRAM) located between CPU cores and the main memory. CPU caches are used to temporarily store data blocks recently accessed by the CPU, so that future read/write operations may be performed only on the cache, i.e. without accessing the RAM. Typically, it takes 3 to 4 cycles for a cache read, while a memory read takes about 250 cycles [18].

As the speed gap between RAM and CPU increases, multiple levels of caches are implemented. Higher-level caches are larger in size (megabytes level) but slower in speed. Some of the caches are dedicated to store data or instructions only, namely data caches or instruction caches, respectively. The cache hierarchy differs among microarchitectures. For example, on Intel Core microarchitecture, each of the two cores on a single die has its own level-one data (L1D) cache (32KB) and instruction cache (32KB). Those two cores also share a unified level-two (L2) cache of 2MB or 3MB.

In symmetric multiprocessing (SMP) systems, different cores may access the same memory location, hence, multiple

copies of the same memory block may be stored in different caches belonging to different cores. As the caches could be modified independently by different cores, data could be inconsistent across multiple copies. To tackle the problem, CPU manufacturers, such as Intel, provide built-in utilities to maintain data consistency between the cache and RAM. However, cache control is usually very limited to the OS and applications.

III. System Design

A. Threat Model

The primary goal of Copker is to defend against memory-based attacks, such as the cold-boot attack [23]. In such attacks, the target computer is physically accessible to the attacker. The attacker could take the following steps to obtain the data in RAM: (1) power off the target computer; (2) pull out the RAM; (3) put it in another machine (fully controlled by the attacker) with a memory dump program; and (4) dump the original contents of RAM to the hard disk of the new machine. To make the attack more effective, the attacker could reduce the temperature of the memory chips to slow down the fading speed of the memory content. Such attacks bypass all access control protections at OS and application levels, since they are essentially at the lowest level (i.e., the hardware).

In this paper, we do not consider OS vulnerabilities, or software attacks against the OS. In particular, we first assume a *trustworthy OS kernel* to prevent attacks at system level, when the attacker has an account (with some privileges) on the target machine. A trustworthy OS ensures basic security mechanisms such as process isolation and hierarchical protection domains. Formally verified OS such as seL4 [31] and PikeOS [5] can be used for this purpose. Meanwhile, unauthorized calls to the private key service are also outside our scope. That is, although an attacker may obtain decryption/signing results, it does not harm the confidentiality of private keys.

Moreover, we also assume that the entire system is safe (i.e., no malicious process stealing secret information) during OS initialization, which is a short time period. In particular, during this period, we derive the AES key-encryption key to be used in Copker, by asking the user to input a password (will be elaborated in Section III-C). We also assume that this password is strong enough to defeat brute-force attacks. After the initialization period, malicious processes may exist in the system (e.g., an attacker gains root privileges and invokes different system calls); however, such processes shall not break the protections by the trusted OS kernel. That is, these attackers can invoke any system call, but the system calls perform as expected.

B. Design Goals and Principles

To defend against cold-boot attacks, our most important design goal is to ensure that sensitive information is never loaded into the RAM. That is, *plain-text* private keys, as well as any intermediate results that might be exploited to expose the keys, are always kept in on-chip CPU caches and registers. Such information should never appear on the front side bus (FSB) or into RAM. On the other hand, to minimize the impact on CPU performance, we only lock the caches when we are using the private keys to decrypt or sign messages. To release unused resources and to protect private keys when they are not used, we employ an existing technique, TRESOR [38], to encrypt private keys with AES, and protect the AES key in CPU registers. In this way, when Copker is not in active mode, the caches can be used normally, so that system performance is not affected. Meanwhile, storing AES-encrypted private keys in RAM is considered safe. The design of dynamic loading also allows us to support multiple private keys simultaneously in Copker.

Copker uses private keys to decrypt and sign messages. To provide decryption and signing services for user-mode processes, and to defend against memory-based attacks, the design of Copker needs to satisfy the following criteria:

1) A fixed address space is allocated and reserved for computing with private keys. During computing, the address space is used only by Copker, so that we can further ensure data in this space are accessed entirely in caches and not written to RAM.
2) All variables, including the plain-text private keys and intermediate variables, are strictly limited within the address space allocated in Criterion 1.
3) The Copker decryption/signing process cannot be interrupted by any other task. Otherwise, the sensitive data in this address space might be flushed to RAM, when cache replacement is triggered by read or write misses (from other tasks).
4) When Copker finishes computing with private key, all sensitive information in this address space is erased. That is, the used cache lines are cleaned deliberately before the cache lines are released.

In Copker, the service is implemented as system functions in Linux kernel. A block of *bytes* are first defined as *static* variables, and then an address space is allocated for these variables. The size of the reserved space is carefully chosen, so that: (a) it is sufficient to hold all variables and data for decryption/signing, and (b) it can be completely filled into the L1D cache. Note that we do not use any heap variable, or store any data in heap, since heap variables are difficult to be limited within the allocated address space. Instead, all sensitive data are stored in stack (as static variables). When a user-mode process requests decryption/signing service, the stack is redirected to the reserved space, before Copker starts to compute with private keys. Therefore, all variables, including the plain-text private keys, of the decryption/signing task are strictly limited within the fixed space. When the task finishes, all the variables in the reserved space are erased.

To satisfy Criterion 3, Copker must be running in an atomic section – all interrupts are disabled during the entire decryption/signing process. Copker enters atomic section before the protected private keys are loaded, and exits atomic section after the cache is erased. On multi-core CPUs, Copker only disables local interrupts, i.e., only interrupts of the core that runs Copker. Besides, Copker relies on the trustworthy OS to prevent tasks on other cores from accessing the reserved space during the decryption/signing process.

Finally, it is very difficult to explicitly obtain consistency status of RAM and caches. Consistency control is performed transparently with the hardware. That is, it is almost impossible to directly check with hardware about whether sensitive

information in cache has been synchronized to RAM. To verify that sensitive information is not written to RAM, we design a validation utility (see Section V-A for details) using the instruction `invd`, which invalidates all cache entries *without* flushing modified data to RAM. After the decryption/signing process, we invalidate all the cache entries, and then check corresponding RAM content. Unchanged contents in RAM imply that cache lines are never flushed to RAM during the decryption/signing process.

C. Private Key Management

Multiple private keys are supported in Copker. When private keys are not used for decryption/signing, they are encrypted by an AES key (i.e., the master key) and stored in hard disks or RAM. When a user-mode process invokes Copker service, the corresponding private key is dynamically loaded, decrypted by the master key, used, and finally erased within the reserved address space.

The Master Key. The master key is derived from a password entered by the user. We assume that the password is strong enough to defeat brute-force attacks. The master key is always protected by TRESOR [38] in four CPU debug registers (in particular, `db0/1/2/3`), to prevent cold-boot attacks. These debug registers are privileged resources which are not accessible from user space and are seldom used in regular applications.

When the operating system boots, a command line prompt is set up for the user to enter the password. The master key is derived and copied to CPU cores. Then, all intermediate states are erased. This vulnerable window lasts for a short period, and only happens early in the kernel space, when the system boots or recovers from the suspend-to-RAM state. Note that before the system suspends, the master key is also erased from registers [38].

With Copker, some hardware debug features become unavailable (e.g., debugging self-modifying codes and setting memory-access breakpoints), because the debug registers are occupied by the master key. Fortunately, the debug register is not the only place to protect the master key against cold-boot attacks. In the literature, other methods have been proposed to store AES keys in different registers, e.g., Amnesia [45] and AESSE [37]. They provide alternative solutions, when the debug registers are necessary for other tasks in the system.

Private Key Loading. When the system boots, the encrypted private keys are pre-loaded into RAM from the hard disk. The private keys are securely generated, and then encrypted by the same master key in a secure environment, e.g., on an off-line trustworthy machine.

To support multiple private keys, and more importantly, to release caches when Copker suspends, a plain-text private key is only decrypted in caches when a decryption/signing request is received. The detailed steps of private key loading are shown in Figure 1: (1) the master key is derived from the user's password and stored in debug registers; (2) the encrypted private keys are loaded into RAM from hard disks; (3) when a decryption/signing request is received, the master key is first written to cache; (4) the requested private key, which is encrypted by the master key, is loaded to cache, and then (5) the private key is decrypted by the master key, to perform private key operations. In the figure, memory locations in shadow indicate encrypted data.

Different from (the original version) of TRESOR, Copker performs AES decryption in caches instead of registers. The plain-text private key will be used to decrypt or sign messages. The operations are performed using CPU cache. Only the decryption or signing results are written to RAM. The cache, with plain-text private key, is erased and then released after decryption/signing. Again, as mentioned above, these steps are in an atomic section, which will not be interruptible by any other operation.

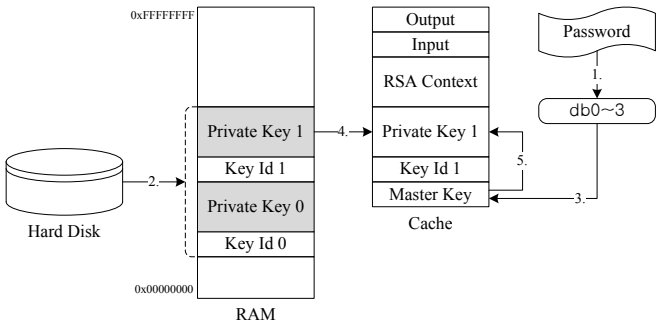

Fig. 1: Dynamic Loading of the Private Keys

D. Copker: Computing with Private Keys without RAM

1) Cache-fill Modes: Before presenting the details of the Copker architecture, we first introduce two cache-fill modes, which play important roles in Copker.

Write-Back Mode. In write-back mode, modified data is not synchronized into the RAM until explicit or implicit write-back operations. This type of memory access is supported by most modern CPUs. It provides the best performance as long as memory consistency is maintained. In the x86 architecture, this mode is enabled only if both *memory type range registers* (MTRRs) and *page attribute tables* (PATs) are set properly. In write-back mode, on cache hits, the execution unit reads from cache lines (read hit) or updates the cache directly (write hit). Meanwhile, on cache misses, cache lines may be filled. The accesses to memory are performed entirely on caches, whenever possible. Write-back-to-RAM operations are only performed when: (1) cache lines need to be evicted to make room for other memory blocks, or (2) cache is explicitly flushed by instructions.

No-fill Mode. The no-fill mode is a processor mode that can be set individually on each core. It is controlled by bit 29 and bit 30 of the `cr0` register on x86-compatible platforms. In this mode, if the accessed memory is write-back type, cache hits still access the cache. However, read misses do not cause replacement (data is read either from another core that holds the newest copy of the data, or directly from the RAM), and write misses access the RAM directly. In short, the cache is frozen, restricting cache accesses only to data that have been loaded in the cache.

2) Computing within the Confined Environment: To satisfy the design criteria presented in Section III-B, we first need

to construct a secure environment that contains all the data/-variables to be used by Coker during the decryption/signing process. The secure environment needs to be entirely stored in the cache, and should not be switched to RAM at any time. This environment should at least include the following elements:

- The master key: the AES mater key copied from debug registers.
- The AES context: the context of the AES master key, including key schedule information.
- The RSA context: the RSA context is initialized by the private key, which is decrypted using the AES context.
- Stack frames: stack frames of functions that compute with private data.
- Input/Output: input and output of the RSA private key operations.

Note that the above environment shall not contain any heap memory. Heap memories are dynamically allocated and the locations are determined by the memory management service of the operating system. Hence, it would be difficult, if not impossible, to restrict heap usage to pre-allocated address and then lock them in cache. Thus, heap memory is not used in Copker's private key operations. In conventional implementations of RSA or other asymmetric cryptographic algorithms, heap memory is primarily used for long integers. Hence, if we are able to define long integers in a static manner, there is no need to use heap memory. In Copker, long integers are handled through static arrays instead of pointers. Therefore, only stack variables are used in our implementation, and use of heap is prohibited.

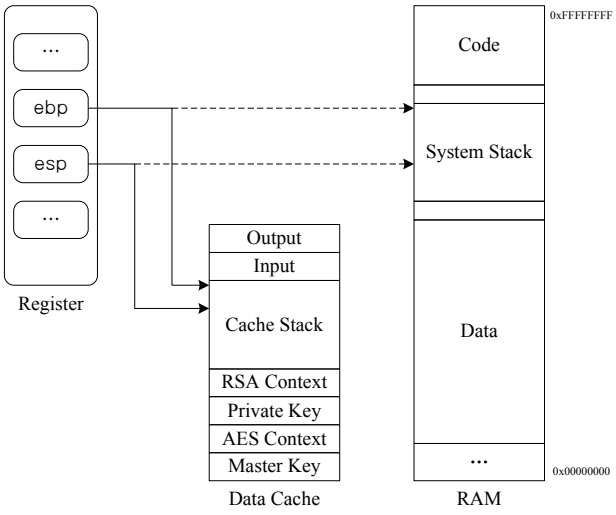

Fig. 2: Stack Switch

However, stack memory location is also not controllable in high level programming languages, such as C. The OS designates the memory locations in the stack for each thread in each ring. Moreover, we cannot prohibit the usage of stack as with heap – without stack, procedure calls in C becomes impossible. To tackle this problem, we temporarily take over the control of stack location, using an approach called *stack switching*, as demonstrated in Fig. 2 (details of the implementation will be presented in the next section). When we compute with private keys, Copker temporarily switches to a customized stack, which, to our expectation, resides in the secure environment defined above.

When Copker is invoked to decrypt or sign a message, the procedure is outlined as follows:

1) The debug registers, which are protected by TRESOR, are loaded to reconstruct the master Key.
2) The AES context is initialized by the master key.
3) The encrypted private key is decrypted using the AES context. Then, we check the integrity of the private key block, by verifying that the public and the private keys match.
4) Using the public and the private keys, the RSA context is initialized.
5) The desired private key operation is performed and the output is fed to the user.
6) The secure environment is erased.

Note that the above operations are all performed on the custom stack in the secure environment as defined above.

3) Securing the Execution Environment: All the sensitive data is confined in the environment described above. We must ensure that this environment only reside in caches when it contains sensitive data. This environment should not be flushed into RAM whenever it is updated. Theoretically, this requirement is perfectly supported by the write-back memory type. However, modern operating systems are complicated: setting the right memory type is only the first step, while more complicate mechanisms are needed to securely "lock" this environment in caches.

Protecting Shared Cache. Higher-level caches (e.g. the L2 cache) are often shared among a set of cores in modern CPUs. When the core running Copker (denoted as core-A) shares a cache with another core (denoted as core-B), the tasks running on core-B may compete for the shared cache with Copker. A memory-intensive task running on core-B may occupy most cache lines of the shared cache. If this shared cache is not exclusive with inner caches (i.e., lower-level caches), Copker's cache lines in the inner caches are also evicted[2]. To prevent this type, the cores sharing caches with core-A are forced to enter *no-fill* mode, so that they cannot evict Copker's caches.

Here we define the *minimum cache-sharing core set* (MCSC set). It is a set of cores that: (1) share with each other some levels of caches that are not exclusive to inner caches, and (2) do not share any cache with cores outside this set. When Copker is running on a core of an MCSC set, all other cores in this set shall switch to no-fill mode.

Atomicity. Multi-tasking is commonly supported in operating systems via context switch, which may be triggered by scheduling, interrupts or exceptions. When context switch is triggered, the states of the running core, including registers, are kept in the RAM. If the task is not resumed very soon, the occupied cache lines may also be evicted. In both cases,

[2]Intel CPUs typically do not implement exclusive caches. Instead, they implement non-inclusive or accidentally inclusive caches for L2 and inclusive caches for L3.

sensitive data may be leaked, if Copker is computing with private keys. To prevent this, Copker works in an atomic section while performing private key operations. In the atomic section, Copker cannot be interrupted by any other task on the same core. The atomic section ensures that all Copker computations are entirely within the confined environment, which is stored only in the cache.

Clearing the Environment. After the decyption/signing task, the plain-text keys and all intermediate states should be erased before Copker leaves the atomic section. Because all the sensitive information is confined in the reserved space, instead of scattered in any memory allocated in heap, it is easier to be erased. We only need to clean the reserved variable space and all registers.

IV. Implementation

Based on the design principles discussed in Section III-D, we implement and integrate Copker into Linux kernel 3.9.2 for 32-bit x86 compatible platforms with SMP (symmetric multiprocessing) support. We have not integrated Copker to formally verified operating systems, such as seL4 and PikeOS. However, such extension is completely feasible. In the prototype, Copker supports 2048-bit RSA, which could be easily extended to support longer keys. Meanwhile, the master key is a 128-bit AES key, which is restricted by the size of debug registers in 32-bit mode, i.e., four 32-bit debug registers.

The interface exported to user space is provided by the `ioctl` system call in a synchronous manner. The `ioctl` system call takes a device-dependent request code to accomplish specific functions in the kernel. In the prototype, we provide 3 functions:

- Get the number of encrypted private keys.
- Perform a private key operation using a specified RSA key, which is identified by `privateKeyId`.
- Get the public part of the key pair identified by `privateKeyId` in plain text.

The structure of Copker API is shown in Figure 3. The sensitive data are completely decoupled from the user-space processes that use the private keys. Moreover, Copker's interface used to exchange data is further encapsulated as an OpenSSL engine [51], making it easy for Copker to be integrated with existing cryptography applications.

Besides the main Copker's implementation, we also implemented a preparation utility, which generates the encrypted private key file on a secure machine. The preparation utility works as follows: (1) the user enters a password, the desired key length, and a unique index of each private key; (2) the master AES key is derived from the password using the same algorithm as Copker; (3) the RSA keys are generated using the OpenSSL library; and (4) the private key blocks are encrypted by the master key, and stored into a file, along with the public parts in plain text.

The prototype is implemented and tested on an Intel Core2 Quad Processor Q8200. As shown in Figure 4, Q8200 contains two cache-sharing core sets, each of which has two cores. Each core has a L1D cache of 32 KB and an instruction cache

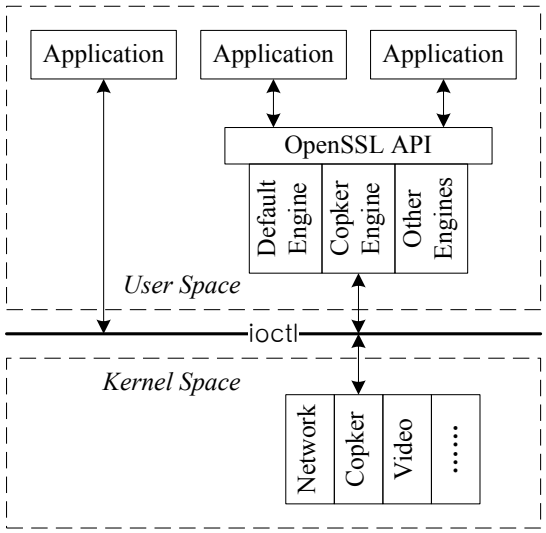

Fig. 3: Copker API Structure

of 32 KB as well. The two cores of the same set share a unified second-level (L2) cache of 2 MB. The L2 cache can be loaded with both data and instructions. L2 cache is non-inclusive, meaning that a cache line in the L2 cache may or may not be in L1 caches. These cache lines on Q8200 compose two separate cache sets: Core 0 and Core 1 share a cache set, while Core 2 and Core 3 share another cache set. Two cache sets are independent.

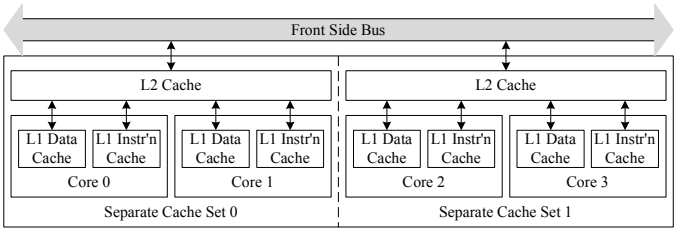

Fig. 4: Cache Hierarchy of Q8200 Processor

A. Cache Control in x86 platforms

As we have introduced, the x86 architecture provides limited cache control utilities to the OS and applications. The cache control registers can be used to control the system-wide cache behaviors. Page-based virtual space caching types and region-based physical space caching types can be set individually. It also provides cache control instructions that can be used to flush all or specified cache lines of a core. In addition, there are instructions to pre-fetch data from system RAM to inner caches. However, none of them could be used to query the status of a specific cache line.

The following cache control utilities are relevant to Copker.

- Control register `cr0`: bits 29 and 30 of `cr0` control the system-wide cache behavior. In normal setting, both bits are cleared, and caching is fully enabled. When bit 29 is cleared and bit 30 is set, the core enters no-fill mode (see Section III-D).

- Instruction `wbinvd`: write back all modified cache lines to the RAM and invalidate the caches. It is worth mentioning that after invalidating the local cache (e.g. L1D), `wbinvd` also signals the corresponding shared cache (e.g. the shared L2) to do the same thing. Note that, `wbinvd` works on the cache set that corresponds to the core – other cache lines outside this cache set are not affected.
- Instruction `invd`: it works in the same way as `wbinvd`, except that the modified data is not written back to the RAM before invalidated. Data in the caches are abandoned.
- Instruction `clflush`: it takes a virtual memory address as operand and invalidates the cache line containing that address. If the cache line contains modified data, the data are flushed to the RAM before the cache line is invalidated.

B. Implementation Details

Execution Environment Definition. `CACHE_CRYPTO_ENV` contains all variables that Copker accesses during the private key operations. This structure is defined in a static manner as shown below:

```
struct CACHE_CRYPTO_ENV {
    unsigned char masterKey[128/8];
    AES_CONTEXT aes;
    RSA_CONTEXT rsa;
    unsigned char cacheStack[CACHE_STACK_SIZE];
    unsigned long privateKeyId;
    unsigned char in[KEY_LENGTH_IN_BYTES];
    unsigned char out[KEY_LENGTH_IN_BYTES];
} cacheCryptoEnv;
```

`CACHE_STACK_SIZE` is 6,400. In the experiments using 2048-bit RSA, the deepest stack that has been used was 5,584 bytes, which means the allocated space is sufficient.`KEY_LENGTH_IN_BYTES` is 256, since the input/output of RSA private key operation must be smaller than the modulus, which is 256 bytes for a 2048-bit RSA. The entire structure occupies 10,292 bytes. To support stronger RSA (i.e., longer keys), more space needs to be allocated. For example, 3072-bit RSA requires 8,028 bytes of `cacheStack` according to our experiments, and the total size of the structure is at least 13,584 bytes.

The size of `cacheCryptoEnv` is much smaller than the size of L1D caches in modern CPUs, which is typically 32K Bytes. Note that `cacheCryptoEnv` is statically allocated in kernel, hence, it is contiguous in both logical and physical memories. 10,292 contiguous bytes are guaranteed to fit in the 8-way set-associative L1D. This is also confirmed by our experiments. In developing the prototype, we have implemented and tested the maximum key length of 4096-bit. Theoretically, we estimate that 7896-bit RSA can be supported if the entire L1D (32 KB) is used.

Stack Switch. In the x86 architecture, register `esp` points to the current stack top, and `ebp` points to the base of the current function's stack frame. The stack operation instructions, e.g., `pushl` and `popl`, implicitly use the base address from the stack segment register (`ss`), plus the operand, to construct a linear address. The current Linux kernel implements flat mode memory, which means that the data and stack segments start from the same virtual address. We can utilize memory area in the data segment as if it was in the stack segment.

`switch_stack(void *para, void *function, unsigned char *stackBottom)` is written in assembly codes to enable stack switching. It first switches from the OS assigned stack to a customized stack with the bottom pointed by `stackBottom`. Then it calls `function` with the parameters pointed by `para`. The code is listed below.

Listing 1: switch_stack() in assembly codes

```
pushl %ebp
movl  %esp,%ebp

movl  16(%ebp),%eax
// eax now points to new stack bottom
movl  %ebp,(%eax)
// save system ebp on the new stack
movl  %ebp,-4(%eax)
// save system esp on the new stack

movl  %ebp,%ebx
// ebx now points to old ebp

movl  %eax,%ebp
movl  %eax,%esp
subl  $4,%esp
// new stack frame created

pushl 8(%ebx)
// parameters for function
call  12(%ebx)
// call function

movl  %ebp,%ebx
movl  (%ebx),%ebp
movl  -4(%ebx),%esp
// now back on the system stack
leave
ret
```

RSA Implementation. Copker's RSA implementation is based on PolarSSL [52], a lightweight and highly modularized cryptographic library. We modified PolarSSL v1.2.5 to eliminate the use of heap memory in its long integer module. Specifically, each long integer is statically allocated 268 bytes, which is the minimum space required to perform a 2048-bit RSA private key operation. In each long integer, 256 bytes are used to store the basic 2048-bit value, and additional 12 bytes are used to carry other auxiliary information. Some of the long integer manipulating functions are modified accordingly. To speed up RSA decryption/signing, PolarSSL implements CRT, sliding windows, and Montgomery multiplication. We change the default value for sliding windows from 6 to 1, to reduce the memory allocation size on the stack with little sacrifice of efficiency.

Filling the L1D Cache. In its active mode, Copker ensures that `cacheCryptoEnv` is in the L1D cache of the local core. In an x86 CPU, when an instruction writes data to a memory location that has a write-back memory type, the core checks whether the cache line containing this memory location is in its L1D cache. If not, the core first fetches it from higher

levels of the memory hierarchy (L2 or RAM) [26] . Taking advantage of this feature, we put `cacheCryptoEnv` to the L1D cache of the core by reading and writing back one byte of each cache line in `cacheCryptoEnv`. This feature only applies to write-back memory type. Therefore, before doing this, we must ensure that `cacheCryptoEnv` has the cache-fill mode of write-back. At the same time, other cores in the same separate cache-sharing core set are configured to no-fill mode, to avoid evicting `cacheCryptoEnv` out of caches.

Atomicity. First, task scheduling is disabled by calling `preempt_disable()`, which disables kernel preemption. By calling `local_irq_save()`, maskable hardware interrupts are disabled as well, so that they will not suspend Copker's execution, which might be exploited (by adversaries) to flush the sensitive information to RAM. Non-maskable interrupts (NMIs) are discussed in Section VI-A. When exiting the atomic section, the two operations are reversed.

Copker with SMP Support. Then there are multiple Copker threads running simultaneously, it is natural that each core is in its own atomic section, and is assigned a `cacheCryptoEnv`. However, cache lines occupied by Copker might be evicted by other cores sharing the same L2 cache, especially when that core is running a memory-intensive task. The result is fatal to Copker: the evicted cache lines, possibly containing sensitive data, are flushed to RAM. To prevent this, only one core in a separate cache set is allowed to execute Copker with write-back cache mode, while all other cores in the set are forced to enter no-fill mode when Copker is running. In the implementation, we define an array of `CACHE_CRYPTO_ENV`, each of which is assigned to a separate cache set. This implies that the maximum number of threads running Copker concurrently is restricted by the number of separate cache sets[3]. For Q8200, Copker can run 2 threads concurrently.

Algorithm 2 demonstrates the main logic of Copker with SMP support. In particular, `SET_CNT` is the number of separate cache sets. Semaphores are used to avoid multiple cores in the same cache-sharing set to execute Copker concurrently, as only one `cacheCryptoEnv` is allocated for each separate cache set. They are implemented with `down()` and `up()`, the PV functions of semaphores in Linux.

At the beginning, the task is restricted in the core where it is running, by setting the thread's affinity to `idCore`. `smp_processor_id(current)` gets the core index of the current task. This avoids inconsistency of `idCore` if the task is scheduled onto another core after Line 1 is executed. Then, `cache_set_id(id)` and `cache_set(id)` return the index and the members of the separate cache-sharing core set, which contains the core identified by `id`, respectively. The information is used to force the cores to enter the no-fill mode.

The function `private_key_compute()` implements the requested private key operations (as described in Section III-D) using the switched stack, whose bottom is pointed by `env->cacheStack+CACHE_STACK_SIZE-4`. Here we subtract 4 from the end of `cacheStack`, because in x86 architecture with 32-bit mode, the stack grows downwards in units of 4 bytes.

[3]Here we refer to real concurrent tasks, not time-sharing concurrency.

Algorithm 2: Copker with SMP support

```
Global Variables: struct CACHE_CRYPTO_ENV
                  cacheCryptoEnv[SET_CNT];
                  semaphore semCopker[SET_CNT];
Input: message, privateKeyId
Output: result
1  idCore ← smp_processor_id(current)
2  set the current thread's affinity to core idCore
3  idCache ← cache_set_id(idCore)
4  if get_memory_type(cacheCryptoEnv[idCache]) ≠
   WRITEBACK then
5  |  exit
6  down(semCopker[idCache])
7  preempt_disable()
8  ℂ ← cache_set(idCore)
9  ℂ ← ℂ\{idCore}
10 for id ∈ ℂ do
11 |  enter_no_fill(id)
12 end
13 local_irq_save(irq_flag)
14 env ← cacheCryptoEnv+idCache
15 fill_L1D(env)
16 env->in ← message
17 env->privateKeyId ← privateKeyId
18 switch_stack(env, private_key_compute,
   env->cacheStack+CACHE_STACK_SIZE-4)
19 clear_env(env)
20 for id ∈ ℂ do
21 |  exit_no_fill(id)
22 end
23 local_irq_restore(irq_flag)
24 preempt_enable()
25 up(semCopker[idCache])
26 return env->out
```

C. Kernel Patch

The Linux kernel is patched to ensure that the sensitive information is stored only in caches and registers. First, the TRESOR patch [38] is installed so that the debug registers that contain the master key are not accessible to other tasks except Copker. The `native_get_debugreg()` and `native_set_debugreg()` system calls accessing debug registers in kernel space are patched, as well as the `ptrace()` system call accessing debug registers in user space. Second, we consider the situations when other tasks interfere with Copker in shared caches. Although direct access to `cacheCryptoEnv` is restricted by the process isolation mechanism of the OS, other tasks in the same separate cache-sharing core set could directly issue cache-related instructions to break our assumption (Criterion 3 in Section III-B). In particular, the following operations on other cores could violate Copker's protection mechanisms, when Copker is in the atomic section.

1) Exit from the no-fill mode by setting `cr0`.
2) Issue `wbinvd` to flush caches that Copker is accessing.

In (1), when the other core exits from no-fill cache mode, malicious tasks can evict Copker's caches by intensive memory

operations. In (2), Copker's caches are directly flushed.

Setting `cr0` and issuing `wbinvd` can only be performed in ring 0, so we only need to patch the corresponding code in kernel. The patch is simple but effective: `wbinvd` and write operations to `cr0` can only be executed if there are no Copker thread running within the same cache-sharing set. This is achieved by requiring the semaphore allocated to the separate cache set. The introduced overhead is negligible, as these operations (e.g., `wbinvd`) are rarely used.

In the Linux kernel for x86 platforms, the instruction `wbinvd` and write operation to `cr0` are both implemented as inline functions, namely `wbinvd()` and `write_cr0()`, in `/arch/x86/include/asm/special_insns.h`. We searched all usages of these two operations in Linux kernel source code, and found that all occurrences strictly invoke `wbinvd()` and `write_cr0()`. The patches to them are similar, hence, we only list the patch to `wbinvd()`. Note that lines marked by "+" indicate code added by the patch, while all other lines belong to the original Linux kernel code.

Listing 2: Kernel patch to wbinvd()

```
static inline void wbinvd(void)
{
+ cpumask_t tempSet,savedSet;
+ int r;
+ unsigned int id;
+ savedSet = current->cpu_allowed;
+ id = smp_processor_id();
+ cpumask_clear(&tempSet);
+ cpumast_set_cpu(id,&tempSet);
+ set_cpus_allowed_ptr(current,&tempSet);

+ r = down_interruptible(semCopker +
      cache_set_id(id));
+ if(r == -EINTR)
+     return;

  native_wbinvd();

+ up(semCopker + cache_set_id(id));
+ set_cpus_allowed_ptr(current,&savedSet);
}
```

Note that there are other operations that might violate Copker's protection mechanism, e.g., setting `MTRRs` to change the memory type of `cacheCryptoEnv`. However, such operation must be executed on the same core as Copker is running on, so it cannot be executed when Copker is in the atomic section. Moreover, we assume `PAT` cannot be changed, as the OS kernel is trustworthy.

Although instruction `clflush` can flush the specified cache lines both in ring 0 and ring 3, it cannot be exploited to break Copker's security protection. First, the user-space code does not have permission to access kernel space, where the sensitive information of Copker is located. Second, Linux kernel does not export any system call that can flush a user-specified memory range. Third, in a trusted kernel, no piece of code would flush `cacheCryptoEnv` directly.

Attackers may flush the translation lookaside buffer (TLB), which is the specific cache for the translation information between virtual and physical addresses. However, but flushing TLB would not affect the corresponding data cache lines for Copker tasks [27].

V. Evaluation

A. Validation

We have designed a mechanism to experimentally prove that the sensitive data in caches are not flushed from caches to RAM. Theoretically, based on the analysis of Algorithm 2, we are sure that `cacheCryptoEnv` in the L1D cache cannot be evicted before it is erased explicitly. However, we would expect to have empirical evidence that we can confirm the data is "locked" in cache. This is considered to be a challenging task, because of the lack of cache control utilities in x86 platform [38, 39]. Memory consistency is automatically maintained by CPU and the RAM controller. However, these are no instruction that can be employed to query the cache line status.

The basic idea of the validation mechanism is as follows: (1) we make a copy of RAM ($\mathcal{C}$) before private key operations; (2) we invalidate cache lines with `invd` after Copker is executed; (3) the data in RAM should not be changed compared with $\mathcal{C}$, unless the cache line has been flushed before `invd` is executed. In practice, we do not make copies of memory. Instead, we first place canary words in `cacheCryptoEnv` in the RAM before any private key operations. After the private key operations, `invd` is issued to invalidate all the modified cache lines, including `cacheCryptoEnv`. Then the copy of `cacheCryptoEnv` in the RAM is checked. If canary words are not crashed, the sensitive data is not written back to the RAM.

Based on Algorithm 2, we add the following steps to validate the correctness of Copker.

1) Fill `cacheCryptoEnv` with canary words, except `in`, `out` and `privateKeyId`, when Copker is initializing. This operation is only performed once. The placed canary words should never be changed afterwards.
2) When entering the atomic section, other cores in the same separate cache-sharing core set execute `wbinvd` before entering no-fill mode. This flushes all the modified data in caches to the RAM on other cores. Then, these cores run without caches.
3) Before calling `private_key_compute()`, Copker executes `wbinvd`. This flushes all the modified data in caches to the RAM on Copker's cores. The `wbinvd` instruction in Steps 2 and 3, is executed to avoid data inconsistency, caused by the `invd` instruction.
4) After `private_key_compute()` returns, Copker flushes out the result by using `clflush` and then executes `invd`. At this time, all the modified data in caches are lost.
5) Check whether canary words are crashed. If so, sensitive data has been potentially leaked into RAM.
6) When leaving atomic section, other cores switch back to normal mode.

It's worth mentioning that caches are flushed in unit of lines, aligned by the cache line size `CACHE_LINE_SIZE`,

which is typically 64 bytes for lower level caches in x86 platform. To avoid flushing data more than `out`, `out` should be aligned by `CACHE_LINE_SIZE`. Therefore, the definition of `out` in `CACHE_CRYPTO_ENV` is changed into the following form:

```
unsigned char out[(KEY_LENGTH_IN_BYTE +
         CACHE_LINE_SIZE - 1)
         / CACHE_LINE_SIZE * CACHE_LINE_SIZE]
__attribute__ ((aligned(CACHE_LINE_SIZE)));
```

We run several Copker threads using the above algorithm concurrently with a memory-intensive program for more than 10 days, and found no cache leakage ever happened. As the above algorithm almost shares the same procedure with Algorithm 2, we are convinced that Copker in Algorithm 2 can effectively protect sensitive data from being flushed into RAM. In the validation, Copker is integrated into the Apache web server to provide RSA decryption services, in response to continuous external HTTPS requests from a client. The HTTPS client runs at the concurrency level of 10. Another memory-intensive program is a infinite loop. In each iteration, it simply requests a 4 MB memory block using `malloc()`, adds up each byte, and then frees the memory block.

Last, we would like to illustrate the slight differences between the validation mechanism and the original Copker approach (presented in Algorithm 2). In Copker, all other cores that share (L2) caches with the Copker core works in no-fill mode. In the validator, the other cores are running without cache, since `wbinvd` is invoked before entering no-fill mode. Furthermore, the validator frequently invokes `wbinvd` and `invd`, both of which are quite expensive. Therefore, although the validator is also capable of keeping sensitive information in caches, we only use it as a validation method. The original Copker prototype is much more efficient than the validator.

B. Performance

We have evaluated the efficiency of Copker and its impact on the overall system performance. We have compared Copker with the modified PolarSSL and the original PolarSSL. The modified PolarSSL is the PolarSSL with modifications by Copker (i.e., static long integer and different sliding window value) but running in the same environment as the original PolarSSL (i.e., the modified PolarSSL does not guarantee that sensitive information only stays in cache). The performance difference between Copker and the modified PolarSSL indicates the loss in performance introduced by adding the protection mechanisms to defeat against cold-boot attacks.

In the following experiments, all these approaches are invoked through OpenSSL engine API to perform 2048-bit RSA decryptions. They use the same RSA keys. The testing machine is a Dell OPTIPLEX 760 PC with an Intel Q8200 processor, which has 4 cores.

Maximum Decryption Operations per Second. We first measure the maximum decryption speed. The client program requests decryption services on each approach, running at different concurrency levels. We record the number of served requests in 10 minutes.

As shown in Figure 5, Copker runs even a little faster than the modified PolarSSL when there are 1 or 2 concurrent threads. This can be explained by the fact that Copker is not affected by scheduling. However, as the concurrency level increases, the modified PolarSSL surpasses Copker: the maximum speed of Copker is only doubled comparing with the single-thread performance, while others are quadrupled. This result is expected: the maximum effective concurrency level of Copker is 2, which is restricted by the number of separate cache sets in the CPU, while the maximum effective concurrency level of other approaches is 4, which is restricted by the number of processor cores.

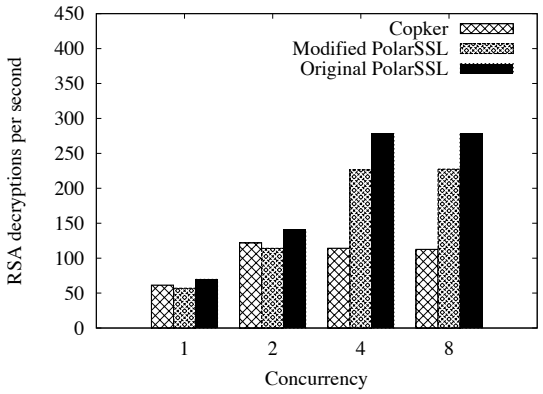

Fig. 5: RSA Decryption Performance

Overall Performance at the Application Level. We then evaluate the performance of Copker, when it is incorporated with other programs. In particular, Copker is integrated into the Apache web server as an RSA component. We measure the throughput of the HTTPS server for varying numbers of concurrent requests.

On the server side, Apache serves a 5 KB html web page under HTTPS protocol with TLSv1.2 using AES128-SHA cipher suite. The client runs on another computer in the 1 Gbps LAN network with the server. We use ApacheBench [49] to issue 10K requests for each approach with various numbers of concurrent clients.

As shown in Figure 6, the expected upper limit of each approach is shown in a solid line. These limits are taken from the maximum value in Figure 5 for each approach. We can see that all of the three approaches are very close to their limits, when concurrency level reaches 200. However, Copker increases slower than others. For example, when the concurrent request number reaches 20, the modified PolarSSL achieved 94.4% of its full capability and the original PolarSSL achieved 91.9%. Meanwhile, Copker can only serve 89.15 requests per second, which is only 73.1% of its maximum speed.

Impact on Concurrent Applications. As Copker forces other cores in the same cache-sharing core set to enter the no-fill mode, the performance of other tasks on these cores may be affected. We use the benchmarking utility SysBench [48] to measure the impact, with a single RSA decryption thread running at different densities. We run SysBench in its CPU mode to do computing-intensive tasks. During the test, the benchmark launches 4 threads to issue 10K requests. Each

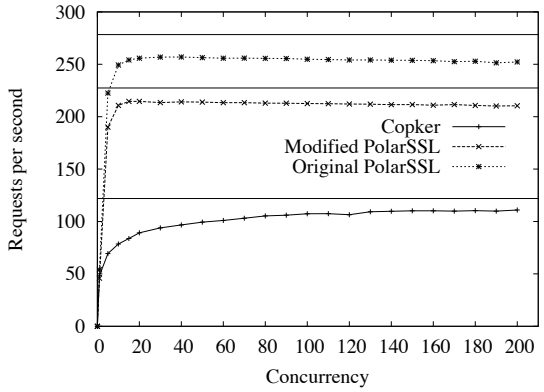

Fig. 6: Apache Benchmark

request consists in calculation of prime numbers up to 30K. The measured score is the average elapsed time for each request. Note that when SysBench spends more time on the task, it indicates that Copker brings higher impacts on the concurrent applications.

In Figure 7, the baseline is measured in a clean environment without any RSA decryption task. At the same decryption frequency, the original PolarSSL performs the best among the three. This is because the original PolarSSL spends less time on each decryption, thereby can spares more computation resources for benchmark tasks. Although Copker performs the worst, the additional overhead is acceptable compared with the original PolarSSL.

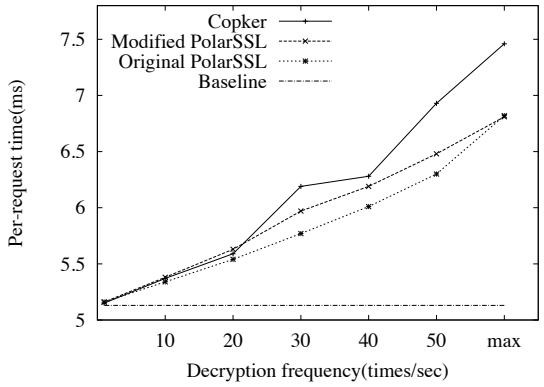

Fig. 7: Impact on Concurrent Applications

C. Discussions

As we have discussed, Copker occupies a CPU core during its decryption/signing operations. In the Q8200 CPU, Copker performs more than 120 decryptions (with 2048-bit RSA key) per second, which is sufficient for many cryptographic applications. In the meantime, CPU cores that share L2 cache with the Copker core is forced into no-fill mode, so that sensitive data in the cache is protected. However, such design also introduces an unavoidable impact on the performance of programs running on the other core, since caches cannot be filled with new data during the computation. Performance degradation becomes severe with newer CPUs, which have a L3 cache shared by all cores. First, shared L3 cache prevents Copker from concurrent operations, since two active Coker threads are mutually exclusive on the L3 cache. Meanwhile, all the other cores have to enter no-fill mode, when a Copker thread is running in active mode.

To mitigate the performance impact in certain applications, we are designing an intelligent scheduling mechanism. The basic idea is to add a *hold* mode to Copker. In the hold mode, Copker collects and holds decryption/signing requests, without performing the operations. Hence, cache lines are not exclusively occupied, and other processes are expected to operate as usual. At time interval t (approximately 0.5 to 2 seconds), Copker switches to *active* mode (possibly multi-threaded) to process all the on-hold requests. As a result, the performance of Copker can be improved by reducing the frequency of switches between the active and suspend modes. Meanwhile, it decreases the performance impact to the tasks running on other cores by reducing the frequency of forcing them into no-fill mode. Note that this solution is only effective in some application scenarios, in which decryption/signing requests arrive at medium frequency, and a small delay could be tolerated for such requests. We plan to further implement this approach in our future work.

VI. Security Analysis

In Section III, we have shown that Copker confines kilobytes of sensitive data in on-chip CPU caches, to defeat against physical attacks, such as the cold-boot attack. Moreover, with intensive experiments, we have validated that our implementation of Copker is correct and no sensitive information is leaked to RAM, as shown in Section V-A. In this section, we provide a theoretical analysis on Copker's resistance against various attacks, such as memory-based attacks, misuses of existing kernel API, and issues with the implementations of asymmetric cryptographic algorithms. We also present the remaining attacking surface, which includes some extremely difficult hardware attacks.

A. OS-Level Attacks

For Copker to operate securely, the following conditions should be held. We will firstly discuss the validity of these conditions, possible attacks and controls. We will then discuss two special scenarios: OS crash, and ACPI states S3 and S4.

- The Copker decryption/signing task cannot be interrupted by other tasks;
- The address space of Copker is not accessed by any other process, when Copker runs in *active* mode, i.e., when Copker is computing with private key;
- The cache of the computing task cannot be influenced by other cores;
- The memory of kernel space cannot be swapped into the disk.

The first condition is partly satisfied, since Copker disables task scheduling and local interrupts, before performing private key operations. However, processor-generated exceptions (invalid opcode, segment not present, etc.) and non-maskable interrupts (NMIs) cannot be disabled through software settings. Processor-generated exceptions can be eliminated through careful programming. On the other hand, NMIs are unavoidable. They are generated in two ways: (1) interprocessor interrupts (IPIs) from advanced programmable interrupt controller (APIC), and (2) external hardware failures. NMI IPIs are widely used to implement software watchdogs to reboot the machine, when the system is stuck. Meanwhile, an adversary could easily trigger external hardware failures, for instance, by overheating the CPU chip. Since NMIs are unavoidable, we need to prevent adversaries from exploiting such NMIs to access sensitive information in cache. That is, the NMI handler needs to be modified to clean `cacheCryptoEnv` in L1D cache immediately after NMI is triggered. Besides, the registers have to be cleared too.

The second condition is mostly ensured with the operating system. In the analysis, we distinguish attackers with different privileges: (1) unprivileged attackers have no way to access other's memory, because Linux kernel enforces process isolation. (2) Privileged attackers may have ways to execute or even modify ring 0 functions – by inserting self-written kernel modules, any code can be executed; by reading `/dev/mem`, any memory in kernel space can be read. Copker should be compiled without loadable kernel module (LKM) and KMEM support to withstand such privileged attacks.

For the third condition, as we have patched the kernel to restrict `wbinvd()` and `write_cr0()` from being called when Copker is running, neither unprivileged nor privileged attackers could influence Copker's cache. Finally, the last assumption is immediately satisfied as the Linux kernel enforces an unswappable kernel space memory.

When OS crashes, the system kernel's memory may be dumped to the disk automatically. This feature is supported by kernel's crashdump (Kdump), which utilizes `kexec` to quickly boot to a dump-capture kernel. As a result, sensitive data in `cacheCryptoEnv` is flushed out to the RAM and contained in the dump, which will be stored on disk. An attacker may take advantage of this system feature, to cause the kernel to crash when Copker is running, by inducing system errors, through either software or hardware. As a countermeasure, `kexec` should not be compiled with kernel, to disallow crash dumps.

Finally, if ACPI state S3 (suspend-to-RAM) or S4 (suspend-to-disk) [25] happens while Copker is in the active mode, we need to ensure that sensitive data cannot be flushed into RAM. Before the ACPI calls (`.prepare` and `.enter`)[4] are issued, the Linux kernel signals all user processes and some kernel threads to call `__refrigerator()`, which puts the caller into a frozen state [50]. Because this call has to wait until Copker leaves the atomic section, nothing sensitive may be written in the RAM or disk.

[4]These calls work in a way similar to BIOS functions.

B. Attacks directly on Copker

We first consider the protection of the password and the master key. Copker employs TRESOR to protect the master key, hence, the master key in debug registers is immune to cold-boot attacks and all analyses of TRESOR also apply to the AES portion of Copker. In particular, during system booting, the kernel (assumed safe) directly reads the password from user, and then derives the master key. All the memory traces during derivation are carefully cleaned, so that both the password and the master key are safe against memory-based attacks. In TRESOR, when the computer wakes up from suspend mode, the administrator has to type in the password again to re-derive the master key (and to access the encrypted hard disk), or he/she has to reboot the computer and enter the password during OS boot-up. This gives attackers more chances if they intend to launch keystroke-logger-based attacks. Unlike TRESOR, we do not need to support master key re-derivation in Copker. Since the AES key is not used to encrypt the hard disk, the computer can still function without the master key. However, if Copker is needed to provide private key services (signing or decryption), the unavailability of the master key can be notified to users through an error code. Meanwhile, the master key needs to be re-deviated by rebooting the machine.

Next, we discuss the security of the asymmetric cryptographic algorithm and its implementation in Copker. At present, our prototype supports raw RSA private key decryption, which is considered to have security breaches [10]. To defeat such attacks, we plan to implement the PKCS#1 standard [28], a provably secure RSA improvement. However, our proof-of-concept prototype has already demonstrated the advantage of computing without RAM, which is the essence of Copker. Meanwhile, other asymmetric cryptosystems could also be implemented in Copker, since the size of CPU cache is capable to handle such operations.

The last issue is the category of side-channel attacks on the implementations of cryptographic algorithms. Theoretically, for a provably secure asymmetric cryptosystem, decryption or signing operations do not leak any information on the private keys [6]. Cache-timing attacks [41] utilize the fact that a spy process running in parallel with the encryption/decryption process (the victim process) can manipulate the shared cache, thus inferring information by observing memory access timings of the victim process. Copker is obviously immune to this type of attacks, since the Copker process only accesses cache during the computations, while no parallel process is allowed on the cache. A recent work found a possible timing attack against the RSA implementation in PolarSSL [2]. This issue was fixed in the most recent release of PolarSSL. Moreover, there are many designs that are resilient to such side-channel attacks, e.g., RSA-blinding [12]. These designs will be employed to improve Copker in the future.

C. Attacks on Hardware

We consider the possibility that the attacker reboots the computer with a malicious booting device (e.g. external USB drive), aiming to image the cache content in a similar way to cold-boot attacks. From the attacker's perspective, if the cache lines were not cleared after rebooting and the attacker knew

the physical address of the corresponding cache line, cache content might have been captured immediately. However, such attack does not work, since internal caches are invalid after power-up or reset [27]. Even though data may remain in cache (depending on the hardware features of the caches), read operation would fetch data from RAM, thereby data in cache is overwritten.

One possible way to fetch data in caches is to directly read the status of transistors in the SRAM cell, or infer the data by side channels, such as electromagnetic field and power consumption. However, although such attacks might be effective to Copker, they are extremely difficult in practice, if not impossible. Theoretically, cryptographic algorithms cannot function without using internal storage (e.g., memory, cache), and CPU cache is amongst the most secure type of storage that could be utilized by CPU. When an attacker is capable of effectively monitoring the hardware at transistor-level, it is extremely difficult to maintain a secure computing environment.

DMA-based attacks [8, 9, 47] are launched from peripherals and are capable of bypassing all the protection mechanisms imposed by the OS. Copker is not designed to withstand this attack. A recent work, BARM [46] monitors bus activity by the performance monitoring units (PMUs) in Intel x86 platform, to detect the abnormal memory access DMA-based attacks from peripherals.

Finally, the joint test action group (JTAG) interface is often used by hardware engineers to debug the chip. The entire state of the CPU can be extracted using the JTAG interface. However, commercial x86 CPUs rarely export JTAG ports [1].

VII. Related Work

Keeping cryptographic keys safe in computer systems is a great challenge, especially when memory is entirely accessible to physical security attackers. AESSE [37], TRESOR [38] and Amnesia [45] improve full disk encryption by storing AES keys in CPU registers, to counter the physical memory attacks such as cold-boot attack [23] and DMA-based attacks [9, 47], both of which bypass the protections of OS completely and enable attackers to access all contents of RAM. These CPU-bound solutions (including Copker) defeat cold-boot attacks effectively, but TRESOR-HUNT [8] shows that they are still vulnerable to DMA-based attacks that actively read and write values to memory on running computer systems. BARM [46], on the other hand, demonstrated a way to detect DMA-based attacks independent of the OS. This work prevents the DMA-based attacks, which are not addressed by Copker.

To protect cryptographic keys against memory-disclosure attacks due to software vulnerabilities [22, 33], K. Harrison and S. Xu [24] suggests only one copy of keys to be kept in memory, and the x86 SSE XMM registers are used to store a 1024-bit RSA private key [40] without the CRT speed-up. PRIME [21], an independently developed approach that was published slightly before Copker, implemented 2048-bit RSA using advanced vector extensions (AVX) [34]. The private key is either symmetrically encrypted in RAM or decrypted only with registers. Some well-chosen intermediate values are stored in RAM, but they would not leak any sensitive information. However, CRT is not enabled due to the limited size of registers, as a result, the decryption/signing operations become less efficient without CRT. In fact, the "one-copy" principle is (followed and) strengthened in both Copker and PRIME: one copy of keys only during computations; otherwise, all private keys are encrypted in memory by another AES key stored in debug registers only. Compared with PRIME, Copker shows better extendability: the large size of cache allows longer private keys and more efficient algorithms, such as CRT-enabled RSA.

White-box cryptography [15] tries to hide a fixed secret key in software binaries, even if the binaries are publicly available, e.g., white-box AES and DES implementations [14, 15]. However, these solutions result in greatly decreased efficiency and do not work effectively for asymmetric algorithms. Our work shows an alternative approach to protect long cryptographic keys, when the machine may be under physical attacks.

The side-channel attack [7, 12] is another threat to cryptographic systems. Firstly, cache-timing attacks [7, 11, 41] are ineffective to Copker, because all cryptographic computations are performed in caches. Note that current AES timing attacks are cache-based [7, 11]. By using CPU AES-NI extension, the AES implementation of TRESOR [38] is free of timing attacks. AES-NI extension is not used in our prototype because Intel Core2 Q8200 doesn't support it. It would be easy to use AES-NI extension in Copker if it is available. Preventing timing attacks will be one of our future work, such as enabling RSA blinding [12] against the attacks to private keys [2].

The cache-as-RAM (CAR) mechanism [35] is employed in most BIOSes, to support stack before the RAM is initialized. To some extent, Copker integrates CAR and TRESOR: cache is used as RAM against cold-boot attacks, and a system-wide AES key is stored in registers as TRESOR to support dynamic multiple private keys. However, straightforward integration is not enough: the execution environment of Copker is more complex than BIOSes and Copker provides cryptographic computing services not only for trusted kernel-mode tasks as TRESOR, but also for untrusted user-mode tasks in Linux. Employing the CAR mechanism, CARMA [53] established a trusted computing base (TCB) with a minimal set of hardware components (i.e., only the CPU), to prevent attacks from compromised hardware.

FrozenCache proposed by J. Pabel [39] is the first attempt to use CAR to mitigate the threat of cold-boot attacks to full disk encryption. This concept is easy to understand, but "the devil is in the details" [29]. FrozenCache only stores the AES key (and its round keys) in caches when a user explicitly actives the frozen mode; otherwise, the cryptographic algorithm is still implemented in RAM and the secret key is also in RAM. That is, FrozenCache uses caches as "pure" storage only, while caches are used in Copker as memory to perform private key operations concurrently with other tasks. Besides, it frozes AES key in the cache by letting the whole CPU enter the no-fill mode to prevent the sensitive information from being flushed into RAM, which is very different from Copker. Therefore, it is very slow for the computer to recover from the frozen mode and harms user experience [39].

VIII. Conclusion and Future Work

Physical attacks on the main memory (RAM) allow attackers, who have physical access to the computer, to extract

RAM contents without any system privilege. Conventional implementations of asymmetric cryptographic algorithms are vulnerable to such attacks, as plain-text private keys are stored in RAM. In this paper, we present Copker, a programming framework for computing with private keys without using RAM. Copker uses CPU caches as RAM to store all private keys and intermediate results, and ensures that sensitive information does not enter RAM. Therefore, Copker is secure against physical attacks on main memory, such as the cold-boot attack.

In Copker, to prevent the sensitive information from appearing on the front side bus and then into RAM: (*a*) a secure environment is designed to store all variables in decryption/signing, (*b*) this environment is placed at reserved memory space, (*c*) the cache-fill mode is carefully configured and the contents of cache is elaborately manipulated to ensure the environment is in caches only, (*d*) the private key operations are performed in the environment, within atomic section to avoid being disrupted by concurrent programs, and (*e*) this environment is completely erased before the reserved space is released. To minimize the performance impact, Copker occupies a limited number of caches only when it is computing with private keys. We implement Cokper, and design a method to verify that the sensitive information is kept in cache only and is never flushed to RAM. Experiment results show that Copker achieves the security goals with good performance and acceptable impact on other programs.

Copker demonstrates a general framework to implement cryptographic algorithms against cold-boot attack and other hardware attacks. In the future, we plan to encapsulate it as an easy-to-use cryptographic module, and to support more algorithms. The large size of cache is capable to support longer private keys and more complicated cryptographic algorithms. Moreover, Copker allows the algorithm to be implemented with high-level programming languages, which makes it easier to extend to other cryptographic algorithms. Last but not least, a random number generater that is immune to cold-boot attacks will be needed, e.g., ECDSA requires secret random numbers to sign messages. It can be implemented by hardware (e.g., the `rdrand` instruction available in Intel Ivy Bridge processors), or a deterministic random bit generator (DRBG) [4], whose sensitive information is also confined in caches.

ACKNOWLEDGEMENT

Le Guan, Jingqiang Lin and Jiwu Jing were partially supported by the National 973 Program of China under award No. 2013CB338001 and the Strategy Pilot Project of Chinese Academy of Sciences under award No. XDA06010702.

REFERENCES

[1] M. Anderson, "Using a JTAG in Linux driver debugging," in *CE Embedded Linux Conference*, 2008.

[2] C. Arnaud and P.-A. Fouque, "Timing attack against protected RSA-CRT implementation used in PolarSSL," in *RSA Conference Cryptographers' Track*, 2013, pp. 18–33.

[3] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Special publication 800-57 recommendation for key management - part 1: General (revised)," National Institute of Standards and Technology, Tech. Rep., 2006.

[4] E. Barker and J. Kelsey, "Recommendation for random number generation using deterministic random bit generators," National Institute of Standards and Technology, Tech. Rep., 2012.

[5] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Formal verification of a microkernel used in dependable software systems," in *28th International Conference on Computer Safety, Reliability and Security*, 2009, pp. 187–200.

[6] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Advances in Cryptology - Crypto*, 1998, pp. 26–45.

[7] D. Bernstein, "Cache-timing attacks on AES," 2005.

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

[9] B. Bock, "Firewire-based physical security attacks on Windows 7, EFS and BitLocker," Secure Business Austria Research Lab, Tech. Rep., 2009.

[10] D. Boneh, "Twenty years of attacks on the RSA cryptosystem," *Notices of the AMS*, vol. 46, no. 2, pp. 203–213, 1999.

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

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

[13] B. Cao and L. Shen, "A survey of VoIP: Now and future," ISGRIN Research Lab, University of Houston, Tech. Rep., 2011.

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

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

[16] T. Dierks and E. Rescorla, "IETF RFC 5246: The transport layer security (TLS) protocol," 2008.

[17] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *13th USENIX Security Symposium*, 2004, pp. 303–320.

[18] U. Drepper, "What every programmer should know about memory," Red Hat, Inc, Tech. Rep., 2007.

[19] A. Freier, P. Karlton, and P. Kocher, "IETF RFC 6101: The secure sockets layer (SSL) protocol version 3.0," 2011.

[20] S. Garfinkel, *PGP: Pretty Good Privacy*. O'Reilly Media, 1994.

[21] B. Garmany and T. Müller, "PRIME: Private RSA Infrastructure for Memory-less Encryption," in *29th Annual Computer Security Applications Conference*, 2013.

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

[23] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten, "Lest we remember: Cold boot attacks on

encryption keys," in *17th USENIX Security Symposium*, 2008, pp. 45–60.
[24] K. Harrison and S. Xu, "Protecting cryptographic keys from memory disclosure attacks," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007, pp. 137–143.
[25] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation, "Advanced configuration and power interface specification," 2006.
[26] Intel Corporation, "Intel 64 and IA-32 architectures optimization reference manual."
[27] ——, "Intel 64 and IA-32 architectures software developer's manual."
[28] J. Jonsson and B. Kaliski, "Public-key cryptography standards (PKCS#1): RSA cryptography specifications version 2.1," RSA Laboratories, Tech. Rep., 2003.
[29] M. Kabay and J. Pabel, "Cold boot attacks: The frozen cache approach," 2009, http://www.mekabay.com.
[30] B. Kaliski, "TWIRL and RSA key size," RSA Laboratories, Tech. Rep., 2003.
[31] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *22nd ACM Symposium on Operating Systems Principles*, 2009, pp. 207–220.
[32] C. Koc, "High-speed RSA implementation," RSA Laboratories, Tech. Rep., 1994.
[33] M. Lafon and R. Francoise, "CAN-2005-0400: Information leak in the Linux kernel ext2 implementation," 2005, http://www.securiteam.com.
[34] C. Lomont, "Introduction to Intel advanced vector extensions," Intel Corporation, Tech. Rep., 2011.
[35] Y. Lu, L.-T. Lo, G. Watson, and R. Minnich, "CAR: Using cache as RAM in LinuxBIOS," 2006.
[36] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.
[37] T. Muller, A. Dewald, and F. Freiling, "AESSE: A cold-boot resistant implementation of AES," in *3rd European Workshop on System Security*, 2010, pp. 42–47.
[38] T. Müller, F. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *20th USENIX Security Symposium*, 2011, pp. 17–32.
[39] J. Pabel, "Frozencache: Mitigating cold-boot attacks for full-disk-encryption software," in *27th Chaos Communication Congress*, 2010.
[40] T. Parker and S. Xu, "A method for safekeeping cryptographic keys from memory disclosure attacks," in *1st International Conference on Trusted Systems*, 2010, pp. 39–59.
[41] C. Percival, "Cache missing for fun and profit," *BSD Conference*, 2005.
[42] B. Ramsdell and S. Turner, "IETF RFC 5751: Secure/multipurpose Internet mail extensions (S/MIME) version 3.2 message specification," 2010.
[43] E. Rescorla, "IETF RFC 2818: HTTP over TLS," 2000.
[44] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.
[45] P. Simmons, "Security through Amnesia: A software-based solution to the cold boot attack on disk encryption," in *27th Annual Computer Security Applications Conference*, 2011, pp. 73–82.
[46] P. Stewin, "A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory," in *16th International Symposium on Research in Attacks, Intrusions and Defenses*, 2013.
[47] P. Stewin and I. Bystrov, "Understanding DMA malware," in *9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2013, pp. 21–41.
[48] SysBench, http://sysbench.sourceforge.net.
[49] The Apache Software Foundation, "Apache HTTP server benchmarking tool," http://www.apache.org.
[50] The Kernelbook Project, "The Linux kernel," http://kernelbook.sourceforge.net.
[51] The OpenSSL Project, "OpenSSL cryptographic library," http://www.openssl.org.
[52] The polarSSL Project, https://polarssl.org.
[53] A. Vasudevan, J. McCune, J. Newsome, A. Perrig, and L. van Doorn, "CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms," in *7th ACM Symposium on Information, Computer and Communications Security*, 2012, pp. 48–52.