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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

Copker: A Cryptographic Engine against Cold-Boot Attacks

Jingqiang Lin, *Member, IEEE,* Le Guan, Ziqiang Ma, Bo Luo, *Member, IEEE,* Luning Xia and Jiwu Jing, *Member, IEEE*

Abstract—Cryptosystems are essential for computer and communication security, e.g., RSA or ECDSA in PGP Email clients and AES in full disk encryption. In practice, the cryptographic keys are loaded and stored in RAM as plain-text, and therefore vulnerable to cold-boot attacks exploiting the remanence effect of RAM chips to directly read memory data. To tackle this problem, we propose *Copker*, a cryptographic engine that implements asymmetric cryptosystems entirely within the CPU, without storing any plain-text sensitive data in RAM. Copker supports the popular asymmetric cryptosystems (i.e., RSA and ECDSA), and deterministic random bit generators (DRBGs) used in ECDSA signing. In its active mode, Copker stores kilobytes of sensitive data, including the private key, the DRBG seed and intermediate states, only in on-chip CPU caches (and registers). Decryption/signing operations are performed without storing any sensitive information in RAM. In the suspend mode, Copker stores symmetrically-encrypted private keys and DRBG seeds 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. We implement Copker with the support of multiple private keys. With security analyses and intensive experiments, we demonstrate that Copker provides cryptographic services that are secure against cold-boot attacks and introduce reasonable overhead.

Index Terms—Cache-as-RAM; Cold-boot Attack; Public-key Cryptography Implementation; Deterministic Random Bit Generator.

1 Introduction

In computer and communication systems, cryptographic protocols are indispensable in protecting data in motion and at rest. In particular, public-key (or asymmetric) cryptography is the foundation of a number of Internet applications. For example, PGP is used to encrypt Emails and verify the identities of senders; SSL/TLS is widely adopted in secure HTTP, anonymous communications, voice over IP and other systems. The security of these systems relies on the confidentiality of private keys. In practice, when the cryptographic engines are loaded, the plain-text keys are usually stored in the main random-access-memory (RAM) of a computer. Although various mechanisms have been proposed for memory protections, the RAM is still vulnerable to physical attacks. When adversaries have physical access to a running computer, they can launch cold-boot attacks [23] to retrieve the contents of RAM chips. Any data stored in RAM, including cryptographic keys, are extracted. The compromised keys could be exploited to decrypt messages, or to impersonate the owners of the keys.

Access control, process isolation and other memory protections at the operating system (OS) level cannot prevent cold-boot attacks, since they are essentially at the lowest level (i.e., hardware). Even though the cold-boot attackers do not have any system privilege in the target machine, they can reboot the machine with removable disks or plug the RAM chips to their own machines, to dump the memory. Existing approaches on memory management (e.g., the one-copy policy [25]) mitigate this problem, by increasing the difficulty to find 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 is disclosed. TRESOR [36] and Amnesia [44] store AES keys and execute encryption/decryption entirely in CPU registers, so that keys are not loaded into the main memory. 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 registers. For example, a 2048-bit RSA private key needs 1152 bytes with Chinese remainder theorem (CRT) speed-up, and the intermediate states need additional 512 bytes at least.

This paper presents a mechanism named *Copker* that **CO**mputes with **P**rivate **KE**ys without **R**AM, to defeat against cold-boot attacks. In particular, we implement RSA and ECDSA, the prevalent public-key cryptographic algorithms, on multi-core CPU systems. Copker also supports an AES-CTR deterministic ran-

- *J. Lin, Z. Ma, L. Xia and J. Jing are with Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, and also with State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences.*
- *L. Guan is with College of Information Sciences and Technology, Pennsylvania State University. This work was conducted when he was at Chinese Academy of Sciences.*
- *B. Luo is with Department of Electrical Engineering and Computer Science, the University of Kansas.*

A preliminary version of this manuscript appeared under the title "Copker: Computing with Private Keys without RAM" in Proc. 21st ISOC Symposium on Network and Distributed System Security (NDSS), 2014.

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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

dom bit generator (DRBG) [5], used in ECDSA signing. During the computations, Copker stores private keys, DRBG seeds and all intermediate states in on-chip CPU caches and registers, and computes entirely on the CPU. Therefore, plain-text keys and sensitive data never appear in 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 as no data are stored in the system heap; (2) the Copker task enters an atomic section, so that it cannot be suspended and the variables are never swapped to RAM; (3) other cores that share caches with the core running Copker tasks are set to the no-fill mode during the computation, so that any task on these cores would not trigger cache replacement; and (4) cryptographic keys are either dynamically loaded into caches or encrypted in RAM, to release the caches when there is no protected computation.

The design goal of Copker is to defend against physical attacks on RAM. Hence, we assume a trustworthy OS kernel – all binaries and processes with root privileges are trusted; underlying software, peripherals' firmware and CPU hardware that the kernel depends on, are also trusted. The prototype system runs in a customized Linux, but it can be ported to a trustworthy OS such as seL4 [28]. The cryptographic algorithms in Copker are written in C language, so they are easier to be extended for more algorithms, compared with assembly codes in [36], [44], [18]. We also designed a validator to verify that no plain-text sensitive data are leaked to RAM under stress tests.

Our contributions are three-fold. (1) Copker is the first solution to perform asymmetric decryption/signing without RAM. We keep cryptographic keys and intermediate states in CPU caches and registers, so that sensitive data never appear in RAM. (2) We implement the designed architecture, and demonstrate its security through security analyses as well as experimental validations. (3) Through intensive experiments, we show that our secure cryptographic services introduce reasonable overhead.

The remainder of this paper is organized as follows. Section 2 presents the background. The design and implementation of Copker are described in Sections 3 and 4, respectively. Section 5 evaluates Copker in term of validity and performance, followed by the security analysis in Section 6. Section 7 surveys the related work and Section 8 draws the conclusion.

2 Background

2.1 The RSA Algorithm

RSA is the most prevalent public-key cryptosystem for both encryption/decryption and signing/verification [42]. A typical RSA private key block is an octuple $(n, e, d, p, q, dp, dq, qinv)$, where (n, e) is the public key, d is the private key, and other variables are private parameters enabling CRT speed-up. CRT makes the computation approximately four times faster than that does not use [29]. The key length (i.e., the length of n) denoted as L, shall be 2048-bit at least [4]. The length of d is also L, while p, q, dp, dq and $qinv$ are $L/2$ in length. Therefore, a 2048-bit RSA private key needs at least $4.5L = 1152$ bytes.

Algorithm 1: RSA Decryption with CRT

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

$t1 = Str2Int(ciphertext)$
$t2 = t1^{dp} \mod p$
$t3 = t1^{dq} \mod q$
$t1 = (t2 - t3) * qinv \mod p$
$t1 = t3 + t1 * q$
$plaintext = Int2Str(t1)$

To execute RSA computations, more memory in addition to the private key block is required. Algorithm 1 shows the RSA decryption with CRT speed-up. The pseudo-code requires at least 3 intermediate variables: $t1$ is L in length, while $t2$ and $t3$ are $L/2$. So, for 2048-bit RSA, at least $6.5L = 1664$ bytes are needed. This pseudo-code shows only the major steps. When we consider the detailed implementation in each step, more memory is needed. For example, to accelerate modular multiplications, Montgomery reduction needs 3 long integers for Montgomery values [34].

2.2 The ECDSA Algorithm

ECDSA [37] is designed for digital signatures only. It works on an elliptic curve $E(\mathbb{F}_p)$ with a base point G. An ECDSA private key is $d \in (1, n-1]$, where n is the order of G. The public key is $P = (x_P, y_P) = d \times G$. The key length (i.e., the length of n) is typically 192, 224, 256, 384 or 521 bits.

Algorithm 2: ECDSA Signing

Input: $digest$, d, G, n
Output: $signature$

$k \leftarrow$ random $\in (1, n-1]$
$(x_k, y_k) = k \times G$
$r = x_k \mod n$
$s = k^{-1} \cdot (Str2Int(digest) + r \cdot d) \mod n$
$signature = Int2Str(r, s)$

ECDSA signing is shown in Algorithm 2. Basically, ECDSA needs less memory than RSA; however, to accelerate the expensive elliptic curve point multiplication, a base-point multiplication precomputation table is usually needed [24]. The random number k needs to be kept secret, which controls the precomputation table lookup; otherwise, attackers could deduce the private key, with a valid signature (r, s) and its corresponding k.

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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

2.3 The AES-CTR DRBG

A DRBG generates deterministic "random" bits with a symmetric cryptographic algorithm and a secret seed. The seed is updated on each generation. The AES-CTR DRBG [5] works as follows. If the seed is kept secret (i.e., the AES key and the counter), the output bits are computationally unpredictable to attackers.

Algorithm 3: AES-CTR DRBG

Input: $length$, key, $cntr$
Output: rnd

```
while l < length do
    rnd = rnd || AESEncrypt(key, ++cntr)
    l = l + 128
t1 = AESEncrypt(key, ++cntr)
t2 = AESEncrypt(key, ++cntr)
(key, cntr) ← (t1, t2)
```

After a certain number of generations, the DRBG is reseeded with entropy input data: the seed is updated by itself and then XORed with the input data.

3 System Design

3.1 Threat Model

The primary goal of Copker is to defend against physical memory-based attacks, such as the cold-boot attack [23]. In such attacks, the target computer is physically accessible to attackers. The attacker takes the following steps to obtain the data in RAM: (1) power off the target computer; (2) pull out the RAM chip; (3) put it in another machine fully controlled by the attacker; and (4) dump the contents of RAM to the attack machine. To make it more effective, the attacker could reduce the temperature of RAM chips to slow down the fading speed of memory contents. The physical attackers are also allowed to probe the front side bus (FSB) that connects the CPU to RAM.

We do not consider OS vulnerabilities, or software attacks on OSes. In particular, we assume a trustworthy OS kernel to prevent attacks at the system level, if the attackers have privileges on the system. The trustworthy OS ensures basic security mechanisms, e.g., integrity and process isolation. Formally verified OSes [28] would be used for this purpose. Such an system implies trusted CPU hardware and peripherals' firmware. Unauthorized calls to the cryptographic service are out of our scope. Attackers might obtain decryption/signing results, but it does not harm the confidentiality of private keys.

Assume that the system is safe (i.e., no malicious process stealing secret information) during its initialization. In this short-time period, we derive an AES key-encryption key to be used in Copker, by asking the user to input a password. This vulnerable window only happens early in kernel space. The password is assumed to be strong enough to defeat brute-force attacks. After the initialization period, malicious processes may exist (e.g., an attacker gains root privileges and invokes system calls); however, such processes shall not break the protections by the trusted OS kernel. That is, the attackers could invoke any system call, but the system calls always perform as expected.

3.2 Design Goals and Principles

To defend against cold-boot attacks, our most important design goal is to ensure that sensitive information never appears on the FSB or in RAM chips. That is, *plain-text* 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. To minimize the impact on CPU performance, we only "lock" the caches when we are using the private keys to decrypt/sign messages or the DRBG seed to generate random bits. To release unused resources and to protect keys when they are not actively used, we employ TRESOR [36], to encrypt them with AES, and protect the AES master key in privileged registers. When Copker is not in its active mode, the caches are used normally, so that system performance is not affected. This design of dynamic loading also enables Copker to support multiple private keys.

The Copker service is implemented as system functions in OS kernel. To provide cryptographic services secure against physical memory attacks, the design of Copker satisfies the following criteria:

1) A fixed address space is allocated and reserved for computing with private keys and secret DRBG seeds. During the computations, the address space is accessed only by Copker, so that we can further ensure data in this space are confined entirely in caches and not written to RAM.
2) All variables are limited strictly in the address space allocated, including private keys, DRBG seeds and intermediate variables.
3) The Copker decryption/signing process cannot be interrupted by any other task. Otherwise, the sensitive data in this 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 keys and seeds, all *sensitive* information in this address space is erased. The used cache lines are cleaned deliberately before they are released.

All sensitive data and variables of the private-key operations are strictly stored and confined within the fixed space. The size of this space is carefully chosen, so that (*a*) it is sufficient to hold all variables and data, and (*b*) it can be completely filled into the level-one data (L1D) cache, typically 32 KB. No data are stored in the system heap, since heap variables are difficult to be limited within a fixed space. When a user-mode process calls decryption/signing services, the stack is

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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

also redirected to the reserved space, before Copker starts to compute with plain-text keys.

For secure public-key algorithms, input and output are allowed to appear as plaintext in RAM, i.e., known to attackers. As for DRBGs, in addition to the seed, the entropy input should be kept secret. In Copker, the entropy input is collected as ciphertext in RAM and decrypted by the AES master key into caches before it is used. Plaintext random bit outputs do not disclose any sensitive states of DRBGs, but they could be exploited to deduce the private keys when used in ECDSA signing. So the DRBG output is protected as sensitive intermediate variables of ECDSA. If an ECDSA precomputation table is used, the access pattern will leak some information about the random bits, and we also need to confine the table access within CPUs.

The Copker task runs in an atomic section: all interrupts are disabled during the computations. It enters the atomic section before any keys or sensitive data are decrypted as plaintext, and exits after the cache is erased. On multi-core CPUs, we disables *local* interrupts, i.e., interrupts of the core that runs Copker.

Finally, it is very difficult to explicitly obtain consistency status of RAM and caches (i.e., whether data in caches are synchronized to RAM or not), because consistency controls are performed transparently by hardware. We design a validation utility using instruction `invd`, which invalidates all cache entries *without* flushing data to RAM. After the decryption/signing process, we invalidate all cache entries, and check the corresponding RAM contents. Unchanged contents in RAM indicate that cache data are not flushed to RAM.

3.3 Key Management

When the private keys and the DRBG seed are not being used, they are encrypted in RAM. When a user invokes Copker services, the requested private key is loaded, decrypted by the master key, used, and finally erased within the reserved space. If the DRBG is needed during this process, the seed is decrypted and updated also in the reserved space, and encrypted again before written back to RAM.

3.3.1 The Master Key

The AES master key is derived from the user password. The 128-bit master key is always protected by TRESOR [36] in four debug registers (in particular, `db0/1/2/3`). The debug registers are privileged resources that are not accessible from user space and seldom used in regular applications. When the system boots, a command-line prompt is set up for the user to enter the password. The master key is derived and copied to each CPU core. Then, all intermediate states are erased carefully.

With Copker, some hardware debug features become unavailable (e.g., debug self-modifying codes),

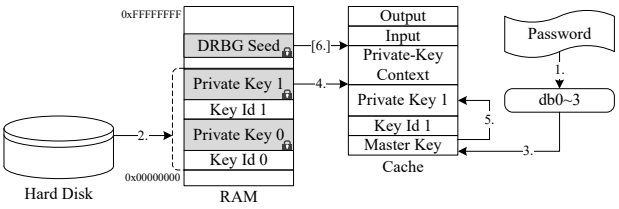

Fig. 1: Dynamic Loading of Private Keys

because debug registers are occupied. However, debug registers are not the only place to protect the master key. The solutions storing AES keys in other registers [44], [35], provide alternative choices, when debug registers are necessary for other tasks.

3.3.2 Private Keys

When the system boots, the encrypted private keys are pre-loaded into RAM from hard disks. These private keys were securely generated, and then encrypted by the same AES key in a secure machine, for example, on an off-line trustworthy computer.

To support multiple private keys, and more importantly, to release caches when Copker suspends, the encrypted private key blocks and DRBG seeds are decrypted in caches only when a decryption/signing request is received. The steps are shown in Figure 1: (1) the master key is derived from the user's password and stored in debug registers; (2) the cipher-text private keys are loaded into RAM from hard disks; (3) when a decryption/signing request is received, the master key is first written to caches; (4) the requested private key is loaded to caches, (5) the private key is decrypted by the master key, to perform private-key operations, and optionally (6) the DRBG seed is decrypted to generate random bits during the private-key operations when necessary. In Figure 1, memory locations in locked shadow indicate encrypted data.

3.3.3 Deterministic Random Bit Generators

DRBGs are used in ECDSA signing, and the DRBG seed is also protected against cold-boot attacks as well as private keys. In particular, when the system boots, random bytes are collected from the kernel entropy pool as an initial "encrypted" DRBG seed in RAM. This initial seed is directly assigned in ciphertext. On each ECDSA signing, the seed is decrypted by the AES master key in caches, and used to generate random bits (see Algorithm 3). Then, the updated seed is encrypted again and written back to RAM.

The DRBG seed is deterministically updated after each generation, and reseeded with entropy inputs after a certain number of generations. Random bytes are exacted from the kernel entropy pool and then decrypted by the master key as entropy inputs. When the system is running, the kernel entropy pool might be disclosed to cold-boot attacks, but the AES master key in privileged registers is kept secure. So the entropy inputs are still unpredictable to attackers.

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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

3.4 A Cryptographic Engine Entirely in Caches

3.4.1 Cache-fill Modes

We introduce two cache-fill modes on Intel CPUs, which play important roles in Copker.

Write-back Mode. In this mode, modified data are not synchronized into RAM until explicit or implicit write-back operations. It is supported by all modern CPUs, and provides the best performance. In Intel CPUs, this mode is enabled when both memory type range registers (MTRRs) and page attribute tables (PATs) are set properly. The accesses to memory data are performed entirely in caches, whenever possible. On cache hits, the core reads from cache lines (read hit) or updates the caches (write hit). Meanwhile, on cache misses, cache lines may be filled. Write-back-to-RAM operations are performed, only if cache lines are (1) evicted to make room for other memory blocks or (2) explicitly flushed by instructions.

No-fill Mode. This mode is enabled individually on each core. In the no-fill mode, if the PAT of accessed memory block is set in the write-back mode, cache hits still access the cache. However, read misses do not cause cache replacement (data are read either from another core that holds the newest copy of the data, or directly from RAM), and write misses access RAM directly. That is, the cache is "frozen", restricting cache access only to data that have been loaded in caches.

3.4.2 Computing within the Confined Environment

We construct an execution environment entirely in caches that contains all data/variables during the protected computations, including these elements:

- The AES mater key copied from debug registers.
- The AES context, including the round keys.
- The private-key context initialized by the plain-text private key bytes.
- The DRBG seed block.
- The stack frames of functions that compute with the sensitive data.
- Input and output of the private-key operations.

The environment shall not contain any system-level heap memory. Heap data are dynamically allocated and the locations are determined by the OS memory management. Hence, it would be difficult, if not impossible, to restrict the heap usage in a pre-allocated address space and then lock them in caches. We reserve a static memory buffer to store all variables of the protected computations. In Copker, the long integer module of public-key cryptographic algorithms which is usually defined in heap, is implemented as static variables or dynamically-constructed ones but within the reserved static buffer (see Section 4.2.1).

C language takes advantages of stack to support function calls. Function parameters and local variables are stored in stack, so the system stack may also contain sensitive data. However, the OS designated memory locations of the stack is uncontrollable. We temporarily take over the stack location control by *stack switching*, as shown in Figure 2. In the Copker system functions, it temporarily switches to a customized stack, which also resides in the secure execution environment as defined above.

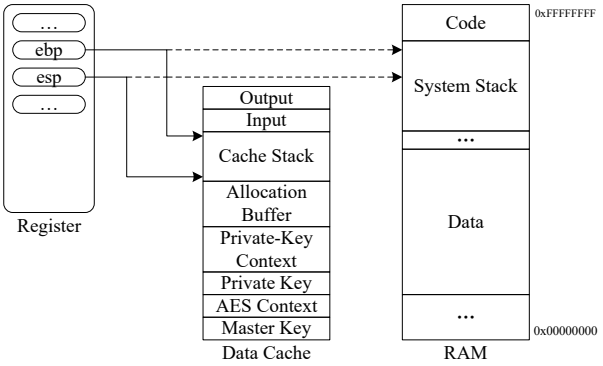

Fig. 2: Stack Switch

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

1) The debug registers 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.
4) The private-key context is initialized, using the plain-text private key block.
5) The desired private-key operation is performed, and, if it is necessary, the DRBG seed is decrypted to generate random bits and then updated.
6) The environment is erased, except the outputs are flushed to RAM. The DRBG seed is re-encrypted and flushed if it is updated.

All above functions are executed on the customized stack in the secure environment. When the DRBG is about to be reseeded with entropy input data, the data are collected *outside* the protected computation, because it uses the kernel entropy pool and the memory access cannot be limited within the allocated space. The number of random bit generations is stored as plaintext, and we prepare the "encrypted" entropy data before starting the protected computation.

3.4.3 Securing the Execution Environment

We must ensure that this environment only resides in caches after it is updated and then contains sensitive variables. It seems that, this requirement has been satisfied by the write-back mode. However, modern OSes are complicated: setting the cache mode is only the first step, while more complicated mechanisms are needed to securely "lock" the environment in caches.

Protecting Shared Caches. Modern CPUs usually implement cache hierarchy with multiple levels, and higher-level caches (e.g., L2/L3 caches) are often shared among a set of cores. For example, as Cores 0 and 1 shares a cache, the tasks running on Core 1

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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

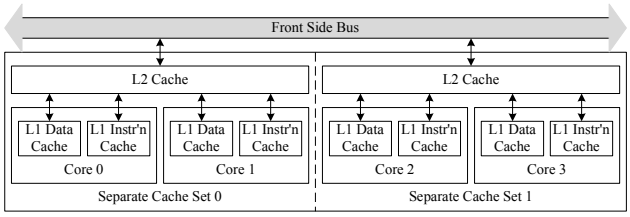

Fig. 3: Cache Hierarchy of Intel Q8200

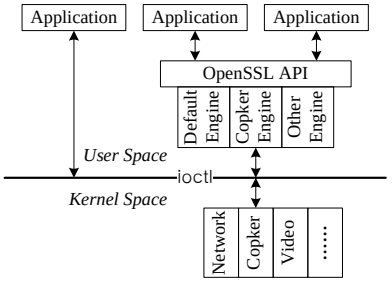

Fig. 4: Copker API Structure

may compete for the shared cache with Copker on Core 0. A memory-intensive task running on Core 1 may occupy most cache lines of the shared cache. If this shared cache is not exclusive with inner (or lower-level) caches, Copker's execution environment in the L1D cache is also evicted. Hence, the cores sharing caches with the Copker core are forced to enter the *no-fill* mode, so that they cannot evict Copker's caches.

We define the *cache-sharing core 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. If Copker is running on a core of a cache-sharing core set, all other cores in this set switch to the no-fill mode.

Atomicity. Multi-tasking is supported in modern OSes via context switch, which is triggered by scheduling, interrupts or exceptions. When context switch is triggered, the states of the suspended task, including registers, are kept in memory. If the task is not resumed soon, the occupied caches may be evicted. In both cases, the data of suspended tasks are leaked to RAM. To prevent this, the private-key operations of Copker work in an atomic section; so it cannot be interrupted by other tasks on the same core.

Clearing the Environment. Before Copker leaves the atomic section, the plain-text keys and all intermediate states shall be erased. Because all sensitive data are confined in the reserved space, instead of scattered in any memory allocated by OSes, it is easier to clear them. We only need to clean the reserved space and all registers, except the output, the encrypted DRBG, and the read-only ECDSA precomputation table.

4 Implementation

We implement and integrate Copker into Linux kernel 3.9.2 for 32-bit x86 compatible platforms with SMP support. In this prototype, Copker supports 2048-bit RSA, 192-bit ECDAS and AES128-CTR DRBGs.

The prototype runs on an Intel Core2 Q8200 CPU. As shown in Figure 3, Q8200 contains two separate cache-sharing core sets, each of which has two cores. Each core has a L1D cache of 32 KB. The two cores of each cache set share a unified L2 cache of 2 MB.

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 Linux kernel. In the prototype, we provide 3 functions:

- Get the number of encrypted private keys.
- Get the public information of a key pair, including the algorithm identifier and its public key.
- Perform a private-key operation using a key pair specified by its index.

Figure 4 shows the structure of Copker API. The API is further encapsulated as an OpenSSL engine, making it easy for Copker to be integrated into existing cryptographic applications.

4.1 Cache Control in x86 platforms

The x86 architecture provides very limited cache control utilities to software. Control register `cr0` is used to control system-wide cache behaviors. Page-based virtual space and region-based physical space cache-fill types can be set individually. It also provides instructions that can be used to flush all or specified cache lines of a core. However, none of them could be used to query the status of a specific cache line.

The following cache control utilities are relevant:

- Control register `cr0`: bits 29 and 30 of `cr0` control system-wide cache behaviors. In normal setting, both bits are cleared, and cache-fill is fully enabled. When bit 29 is cleared and bit 30 is set, the core enters the no-fill mode (see Section 3.4).
- Instruction `wbinvd`: write back all modified cache lines to RAM and invalidate the cache lines. After invalidating the local cache (e.g. L1D cache), `wbinvd` also signals the corresponding shared cache (e.g. shared L2 cache) to do the same thing. Note that, `wbinvd` works on the cache set that corresponds to the core – other cache lines outside this set are not affected.
- Instruction `invd`: it works in the same way as `wbinvd`, except that the modified data are not written back to RAM before invalidated. Data in the caches are abandoned.
- Instruction `clflush`: it takes a virtual memory address as operand and flushes the (modified) data in the corresponding cache line into RAM. Then, the cache line is invalidated.

4.2 Implementation Details

4.2.1 Cryptographic Algorithms

RSA. Copker's RSA implementation is based on PolarSSL v1.2.5 [51], a lightweight cryptographic library.

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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

We eliminate the heap usage in its long integer module. Each long integer is *statically* allocated 268 bytes: 256 bytes store the basic 2048-bit value, and 12 bytes for other auxiliary information. To speed up RSA decryption/signing, PolarSSL implements CRT, sliding windows, and Montgomery multiplication. We change the default value of sliding windows from 6 to 1, to reduce the memory allocation in stack with little sacrifice of efficiency.

ECDSA. ECDSA is implemented on PolarSSL v1.3.5, for v1.2.5 does not support ECDSA. This improved version also introduces built-in dynamic memory functions, independent of the OS dynamic memory library. These functions are used for long integers and require a fixed memory range that is specified in the reserved space. PolarSSL uses a 16-point precomputation table to speed up base-point multiplications, and entries of the table are constructed one by one on the fly as it is used in ECDSA signing. We modify it to construct the whole table when Copker is initializing; and the table is loaded into the L1D cache as a whole, to defeat against FSB probing attacks (see Section 6.1).

DRBG. The AES-CTR DRBG is also borrowed from PolarSSL. The DRBG seed is 48 bytes, consisting of a 256-bit AES key and a 128-bit counter. It reseeds after 10,000 times of function calling. The system function `get_random_bytes()` returns entropy data to initialize the seed and to reseed the DRBG.

4.2.2 Execution Environment Definition

`CACHE_CRYPTO_ENV` contains all variables that Copker accesses during the private-key operations. This data structure is defined in a *static* manner as below.

```
struct CACHE_CRYPTO_ENV {
  unsigned char masterKey[128/8];
  AES_CONTEXT aes;
  union {
    RSA rsaCtx;
    ECDSA {
      ECDSA_KEY ecdsaKey;
      unsigned char mallocBuffer[DMEM_SIZE];
    } ecdsaCtx;
  } pkcontext;
  DRBG drbg;
  PRECOMPUTATION_TABLE table;
  unsigned char cacheStack[CSTACK_SIZE];
  unsigned long privKeyId;
  unsigned char input[MAX_IN_LENGTH];
  unsigned char output[MAX_OUT_LENGTH];
} cacheCryptoEnv;
```

`CSTACK_SIZE` is 6,400, sufficient in all our experiments. In the 2048-bit RSA experiments, the deepest stack that has been used is 5,584 bytes, while it is 1,376 bytes in 192-bit ECDSA. Input and output of RSA private-key operations are both 256-byte. The input of ECDSA signing is typically 32 or 20 bytes (i.e., a SHA1 or SHA256 digest), and the output is 48-byte. In the 192-bit ECDSA experiments, 5,120 bytes of `mallocBuffer` was used for the long integer module in ECDSA, not including the precomputation table `table`. Finally, `table` needs about 1,344 bytes.

The structure occupies about 15 KB, for both 2048-bit RSA and 192-bit ECDSA. To support stronger keys, more memory are needed. For example, 3072-bit RSA requires 8,028 bytes of `cacheStack` in our experiments, while 256-bit ECDSA requires 5,632 bytes of `mallocBuffer` and about 1,600 bytes of `table`.

The size of `cacheCryptoEnv` is smaller than the size of L1D caches in Intel CPUs, typically 32 KB. Note that `cacheCryptoEnv` is statically allocated in kernel, hence, it is contiguous in both logical and physical memories. Contiguous 15 KB are guaranteed to fit in the 8-way set-associative L1D cache, without any conflict. This is confirmed in our experiments. In developing the prototype, we have tested the maximum RSA key length of 4096-bit and 256-bit ECDSA.

4.2.3 Filling the L1D Cache

In an x86 CPU, when an instruction writes data to a memory location that is in the write-back mode, the core checks whether this location is in its L1D cache. If not, the core first fetches it from higher levels of the memory hierarchy (i.e., L2/L3 caches or RAM) [26]. Taking advantage of this feature, we put `cacheCryptoEnv` into the L1D cache by simply reading and writing back one byte of each cache line. Before doing this, we ensure that `cacheCryptoEnv` is in the write-back mode.

4.2.4 Stack Switch

In x86 platforms, 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, plus the operand, to construct a linear address. Linux kernel implements flat-mode memory, which means that the data segment and the stack segment start from the same virtual address. We utilize memory area in the data segment as if it was in the stack segment.

4.2.5 Atomicity

First of all, task scheduling is disabled by calling `preempt_disable()` that disables kernel preemption. By calling `local_irq_save()`, maskable interrupts are disabled as well, so they will not suspend Copker's execution. When Copker exits the atomic section, two operations are reversed. System management interrupts (SMIs) and non-maskable interrupts (NMIs) are discussed in Section 6.2.

4.2.6 SMP Support

When the core in a cache-sharing core set is running Copker, all other cores in the set are forced to enter the no-fill mode. It implies that the maximum number of threads running Copker concurrently, is restricted

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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

by the number of separate cache sets. Here, we refer to real concurrent tasks, not time-sharing concurrency. Intel Q8200 can run two concurrent Copker threads.

Algorithm 4: Copker with SMP Support

```
   Variable: cacheCryptoEnv[CSET_CNT],
             semaphoreCopker[CSET_CNT]
   Input: message, keyId
   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  env ← cacheCryptoEnv+idCache
5  if get_memory_type(env) ≠ WRITE_BACK then
6  |  exit
7  end
8  down(semaphoreCopker[idCache])
9  preempt_disable()
10 local_irq_save(irq_flag)
11 ℂ ← cache_set(idCore)\{idCore}
12 for cid ∈ ℂ do
13 |  enter_no_fill(cid)
14 end
15 fill_L1D(env)
16 env->(input, privKeyId) ← (message, keyId)
17 switch_stack(env, private_key_compute,
   env->cacheStack+CACHE_STACK_SIZE-4)
18 clear_env(env)
19 for cid ∈ ℂ do
20 |  exit_no_fill(cid)
21 end
22 local_irq_restore(irq_flag)
23 preempt_enable()
24 up(semaphoreCopker[idCache])
25 return result ← env->output
```

Algorithm 4 lists the main logic of Copker with SMP support. In particular, `CSET_CNT` is the number of separate cache sets. Semaphores are used to avoid multiple cores in the same cache set to run Copker concurrently, as only one `cacheCryptoEnv` is allocated for each 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`. It avoids inconsistency of `idCore` if the task is scheduled onto another core after Line 1 is executed. `cache_set_id(id)` and `cache_set(id)` return the index and the members of the cache-sharing core set that contains the core identified by `id`, respectively. Then, these cores enter the no-fill mode.

`private_key_compute()` executes the private-key operations using the switched stack, pointed by `env->cacheStack+CSTACK_SIZE-4`. We subtract 4 from the end, because in 32-bit x86 platforms, the stack grows downwards in units of 4 bytes.

4.3 Kernel Patch

Linux kernel is patched to ensure that sensitive data are only in caches and registers. TRESOR patch is firstly applied [36], so the debug registers that contain the master key are unaccessible to other tasks except Copker. `ptrace()` accessing debug registers in user space is patched, as well as `native_get_debugreg()` and `native_set_debugreg()` that access debug registers in kernel space. Second, although direct access to `cacheCryptoEnv` is restricted by process isolation of the OS, other tasks in the same cache-sharing core set could issue cache instructions to violate Copker's protections, when Copker is in the atomic section: (1) exit from the no-fill mode by setting `cr0`; and (2) issue `wbinvd` to flush caches that Copker is accessing.

Setting `cr0` and issuing `wbinvd` are only executed with ring 0 privileges, so we only need to patch the corresponding kernel codes: write operations to `cr0` and `wbinvd` can only be executed if there is no Copker thread running within the same cache set. The introduced overhead is negligible, as these operations are rarely used.

In Linux kernel for x86 platforms, instruction `wbinvd` and write operations to `cr0` are implemented as inline functions, `wbinvd()` and `write_cr0()`, in `/arch/x86/include/asm/special_insns.h`. We searched all usages of these operations in Linux kernel source codes, 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()` as below. The lines marked by "+" indicate codes added by the patch, while other lines belong to the original Linux kernel codes.

Listing 1: Kernel patch to wbinvd()

```
static inline void wbinvd(void)
{
+ cpumask_t tempSet,savedSet;
+ int idCore;
+ savedSet = current->cpu_allowed;
+ idCore = smp_processor_id();
+ cpumask_clear(&tempSet);
+ cpumast_set_cpu(idCore,&tempSet);
+ set_cpus_allowed_ptr(current,&tempSet);
+ if(-EINTR == down_interruptible(
+   semaphoreCopker + cache_set_id(idCore));
+     return;
  native_wbinvd();
+ up(semaphoreCopker + cache_set_id(idCore));
+ set_cpus_allowed_ptr(current,&savedSet);
}
```

There are other operations that might violate Copker's protections, e.g., setting `MTRR` or PATs to change the cache mode of `cacheCryptoEnv`. `MTRR` operations must be executed on the same core as Copker is running on, so it is impossible during the atomic Copker computations. And PATs cannot be changed, because the OS kernel is (assumed to be) trustworthy. Besides, attackers may flush the translation lookaside

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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

buffer (TLB), the specific cache for the translation between virtual and physical addresses; but flushing TLB does not affect data caches [27].

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

5 Evaluation

5.1 Validation

We 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 4, we ensure that `cacheCryptoEnv` in the L1D cache cannot be evicted before it is erased explicitly. However, we expect to have empirical evidences that the data are locked in caches. This is considered to be a challenging task [39], [36], because memory consistency is automatically maintained by CPUs and the RAM controller in x86 platforms, and these is no instruction that queries the cache line status.

The basic idea of our validator is as follows. We first place canary words in `cacheCryptoEnv` in the RAM before any private-key operation. After the private-key operation, `invd` is issued to invalidate all the modified cache lines of `cacheCryptoEnv`, without flushing them to RAM. Then the copy of `cacheCryptoEnv` in RAM is checked. If all canary words remain, no data are written to RAM.

Based on Algorithm 4, we add the steps as below to validate the correctness of Copker:

1) Fill `cacheCryptoEnv` with canary words, except `drbg`, `table`, `privKeyId`, `input` and `output`, when Copker is initializing. This operation is only executed once.
2) When Copker starts the atomic section, other cores in the same cache-sharing core set execute `wbinvd` before entering the no-fill mode. It flushes all modified data in other cores' caches to RAM. Then, these cores run without caches.
3) Before calling `private_key_compute()`, Copker executes `wbinvd`. It flushes all modified data in caches to the RAM on Copker's core.
4) After `private_key_compute()` returns, Copker flushes the result and the updated DRBG by using `clflush`, and then executes `invd`. At this time, all other modified data in caches are lost. Instruction `wbinvd` in Steps 1 and 2, is executed to avoid data inconsistency caused by `invd`.
5) Check whether canary words are crashed, when leaving the atomic section. If all canary words keep unchanged, it is verified that no data are flushed to RAM; otherwise, sensitive data may have been leaked into RAM.

Caches are flushed in units of cache lines, typically 64-byte for the L1D caches. To avoid flushing data more than `output` and `drbg`, the definition of `output` is changed as below, and `drbg` is done similiarly.

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

We ran Copker services using the above algorithm concurrently with a memory-intensive program for more than ten days, and found no cache leakage. That is, all canary words keep unchanged during this validation experiment. As the above algorithm almost shares the same procedure with Algorithm 4, we are convinced that Copker effectively protect sensitive data from being flushed into RAM. In the validation, Copker is integrated into an Apache web server to provide RSA/ECDSA decryption/signing services, in response to continuous HTTPS requests from a client at the concurrency level of 10. The memory-intensive program is an infinite loop. In each iteration, it requests a 4 MB block using `malloc()`, adds up each byte, and frees the memory.

Although the validator is also capable of keeping sensitive information in caches, we only use it as a validation method. The Copker prototype is much more efficient. In the final design, all other cores that share caches with the Copker core works in the no-fill mode. In the validator, other cores are running *without* any cache, since `wbinvd` is invoked before entering the no-fill mode. Moreover, the validator invokes `wbinvd` and `invd`, both of which are expensive.

5.2 Performance

We evaluated the efficiency of Copker and its impact on the overall system performance. It is compared with the *modified* PolarSSL and the *original* one. The modified version is the PolarSSL with the modifications by Copker, including (*a*) long integers without heap and modified sliding window values in RSA, and (*b*) long integers with built-in dynamic memory and the precomputation table in ECDSA, but it runs in the same environment as the original version. The difference between Copker and the modified PolarSSL indicates the performance loss from the protections.

In these experiments, all approaches are invoked through OpenSSL engine API to perform 2048-bit RSA decryption or 192-bit ECDSA signing. They use the same RSA (or ECDSA) key. The testing machine is a Dell OptiPlex 760 PC with one Intel Q8200 CPU.

5.2.1 Maximum Private-key Operations per Second

We measure the maximum private-key operations speed. The client program requests services on each

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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

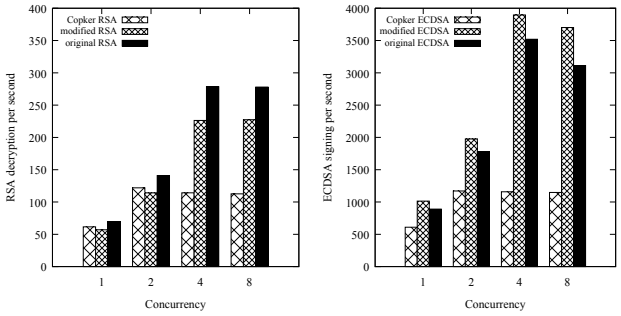

Fig. 5: RSA and ECDSA Performance

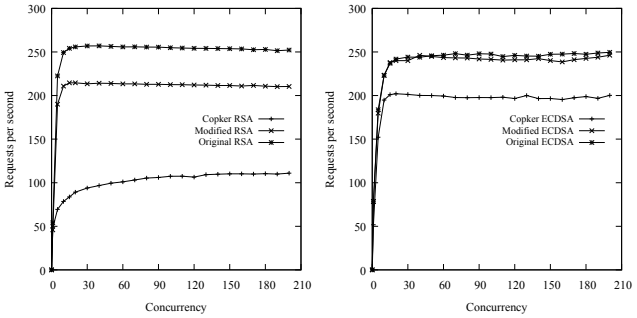

Fig. 6: Apache Benchmark

approach, running at different concurrency levels. We record the number of served requests in 10 minutes.

As shown in Figure 5, Copker runs comparably to PolarSSL when there are 1 or 2 concurrent threads. However, as the concurrency level increases, PolarSSL surpasses Copker: compared with the single-thread case, the maximum speed of Copker is only about doubled, while others are quadrupled. This result is expected: the maximum effective concurrency level of Copker is 2, which is restricted by the number of cache sets in the CPU, while this level of others is 4, restricted by the number of CPU cores. Copker RSA runs a little faster than the modified PolarSSL, because it is not affected by task scheduling; due to the reduced sliding windows of RSA, the modified PolarSSL is less efficient than the original one. On the contrary, an ECDSA task is much more lightweight than RSA, so disabled task scheduling does not bring remarkable benefits. The modified ECDSA is better than the original one, because each thread keeps a separate buffer as its long-integer space while the original PolarSSL v1.3.5 needs to synchronize these threads on the OS dynamic memory. We also found that, when there are many concurrent tasks (e.g., 16 threads), the efficiency of PolarSSL v1.3.5 decreases, probably due to its built-in dynamic memory option.

5.2.2 Overall Performance at the Application Level

We evaluate the performance of Copker, integrated into an Apache web server as the HTTPS private-key engine. Apache serves a 5 KB web page under HTTPS with TLSv1.2. The TLS cipher suit is RSA-AES128-SHA or ECDHE-ECDSA-AES128-SHA. The client runs on another computer in 1Gbps LAN with the server. ApacheBench [49] issues 10K requests with various levels of concurrent requests, and we measure the HTTPS server throughput.

The HTTPS throughput is shown in Figure 6. The upper limit due to the process of network packets and protocols, is about 250 requests per second. When the original PolarSSL (RSA or ECDSA) and the modified ECDSA are used, their results are very close, about 250 requests completed per second. The modified version and the Copker version of RSA reach closely to their limits in Figure 5, as the concurrency level increases. Although Copker finishes ECDSA signing more than 500 times per second, its HTTPS throughput is less than the upper limit (only about 200 requests per second) due to the overhead of Copker's protections.

5.2.3 Impact on Concurrent Applications

As Copker forces other cores in the same cache-sharing core set to enter the no-fill mode, the performance is affected. We use SysBench [48] to measure the impact, as a single Copker thread is running at different densities. Figure 7 shows the results of SysBench in its CPU mode. The benchmark launches 4 threads to issue 10K requests. Each request consists in calculation of prime numbers up to 30K. The score is the average time for each request. When SysBench spends more time on the task, Copker brings higher impact on the concurrent applications. In Figure 7, the baseline is measured in a clean environment without any task. At the same request frequency, the original PolarSSL performs the best among the three. The original PolarSSL spends fewer resources on cryptographic tasks, thereby spares more resources for benchmark tasks. Although Copker performs the worst, the additional overhead is acceptable – compared with the original PolarSSL, about 10–35% more time for each SysBench task depending on different cryptographic algorithms and densities.

We also ran SysBench in the memory mode. Table 1 shows the throughput of memory access (four benchmark threads, one KB each read or write operation for 3 GB data), as the cryptographic engine is called at its

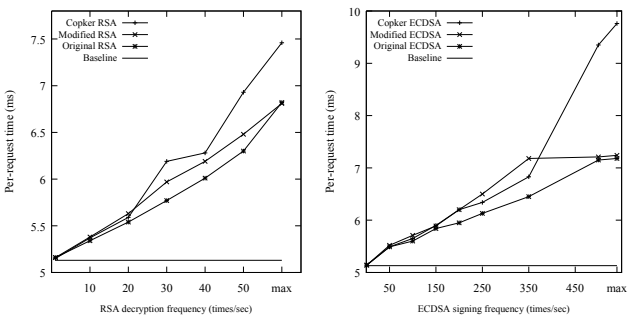

Fig. 7: CPU Impact on Concurrent Applications

TABLE 1: The Throughput of Memory Access

	Write (MB/s)	Read (MB/s)
Baseline	957.69	1276.76
Copker RSA	302.19	352.26
Modified RSA	520.19	693.74
Original RSA	500.75	659.12
Copker ECDSA	614.90	659.23
Modified ECDSA	725.46	1020.51
Original ECDSA	739.36	1022.25

maximum speed by one thread. The baseline results are measured without any cryptographic engine. It is shown that, either for RSA or ECDSA, Copker brings the most significant impact. The modified PolarSSL performs the best for RSA (i.e., brings the least impact to memory access), while the original version does the best for ECDSA. The impact of memory access by Copker is acceptable – compared with the best one, the throughput under Copker is about 51–58% for RSA (or about 64–84% for ECDSA).

5.3 Performance Optimization

During the protected computations, CPU cores that share caches with the Copker core are forced into the no-fill mode. Such design brings performance impacts: the caches of these core cannot be filled with new data during the Copker computations, and the number of concurrent Copker tasks is limited. It becomes severe on newer CPUs, which usually have a L3 cache shared by all cores. Such platforms will not run concurrent Copker tasks. Meanwhile, one Copker thread on Intel Q8200, performs more than 60 2048-bit RSA decryption per second or 500 192-bit ECDSA signing, which is sufficient for many applications.

To mitigate this performance impact, Copker can work with an intelligent scheduling scheme. The idea is to add a hold mode. In this mode, Copker still runs as a service to collect and hold decryption/signing requests from applications, but not performing the private-key operations. It periodically switches to the active mode to process all on-hold requests. Therefore, the performance will be improved by reducing the switch between the active and suspend modes. It also decreases the performance impact to the tasks on other cores by reducing the frequency of forcing them into the no-fill mode. This scheme is only effective in some scenarios, where Copker requests arrive at medium frequency and a small delay is tolerated.

5.4 Applicability

The prototype system is implemented on Intel Q8200 CPUs and a customized Linux kernel, but the design is applicable to other platforms. Firstly, Copker depends on two cache-fill modes, i.e., write-back mode and no-fill mode. These modes are supported on most (and almost all recent) Intel CPUs, and lots of ARM and AMD CPU chips support similar controls [1], [31]. Secondly, the Copker design requires privileged registers to protect the master key, which are also available on ARM and AMD platforms [44], [13] as well as on Intel CPUs. Finally, the protected computation runs in an atomic section, and atomicity is a general function of commodity OSes. So it is practicable to integrate Copker into different platforms as common security services. Moreover, since it accesses only caches during the computations, such services offer the potential to prevent cache-based side channels in the virtualization environment [57], [41], which will be our future work.

6 Security Analysis

6.1 Attacks directly on Copker

We consider the protection of the password and the AES master key. Firstly, Copker employs TRESOR to protect the master key, hence, all analyses of TRESOR also apply to (the AES portion of) Copker. On system booting, the OS kernel reads the password from users to derive the master key. All the memory traces must be carefully cleaned. In TRESOR, when the computer wakes up from the suspend mode, the administrator may type in the password again to derive the master key (and to access the encrypted hard disk); or, reboot the computer. This option gives attackers chances to compromise the password (i.e., the master key) if they launch keystroke-logger attacks, so Copker does not support such master key re-derivation. Since the AES master key is not used to encrypt disks, the computer still functions without it. If Copker is then invoked to provide services, the unavailability of the master key is notified to users via an error code. The master key has to be re-deviated by rebooting the machine.

A skillful attacker may launch bus-probing attacks, to monitor the FSB connecting the main memory and CPUs. As shown before, the plain-text sensitive memory data in `cacheCryptoEnv` is only generated inside the L1D cache and never appears in the FSB. In ECDSA signing, the precomputation table is loaded into the L1D cache as a whole; otherwise, if the table entries are loaded into caches one by one across the FSB as $k \times G$ is being computed, such bus-probing attacks could infer some information about k.

The OS entropy pool is not always random enough; sometimes, its outputs might be predictable [15], [16]. In Copker, although the OS-provided entropy is used to initialize and reseed the DRBG, the actual inputs appear only after the entropy data are decrypted with the AES master key. If the master key is well-protected, these inputs are computationally unpredictable and random to attackers.

The last attacks are side channels on cryptographic engines. Exploiting the fact that accessing cached data is about two orders of magnitude faster than those in RAM [17], various cache-based timing attacks could detect cache hits and misses during the computations

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.2016.2631548, IEEE Transactions on Dependable and Secure Computing

and then deduce the keys [30], [55], [7], [9]. Copker is immune to such attacks, because it accesses only caches during the computations and no concurrent task is allowed to control the shared caches. Another timing side channel [2] on the RSA library of PolarSSL v1.1.4, has been fixed in the recent versions. Other side channels are built by analyzing electromagnetic fields [19], power [38], ground electric potential [20] or acoustic emanations [21]. There are algorithm designs against these side channel attacks, e.g., RSA blinding [10] and ECC randomization [14]. We will integrate these designs into Copker in the future.

6.2 OS-Level Attacks

For Copker to operate securely, the following conditions shall be held: (1) the Copker decryption/signing execution cannot be interrupted by other tasks; (2) the reserved address space is not accessed by any other process; (3) the cache of Copker tasks cannot be influenced by other cores; and (4) the memory of kernel space cannot be swapped into hard disks.

The first condition is partly satisfied, since Copker disables task scheduling and local interrupts, before private-key operations. However, NMIs, SMIs and processor generated exceptions (e.g., segment not present, invalid opcode) cannot be disabled by software settings. Processor generated exceptions can be eliminated through careful programming; but NMIs and SMIs are unavoidable. Therefore, we need to prevent adversaries from exploiting SMIs or NMIs to access sensitive information in caches. That is, SMI/NMI handlers need to be modified to clean `cacheCryptoEnv` in L1D caches (and registers) immediately after these interrupts are triggered.

The second condition is mostly ensured with OSes. Unprivileged processes cannot access others' memory, because the OS enforces process isolation. Privileged attackers may have ways to access Copker's memory – by inserting self-written kernel modules, any ring 0 code can be executed; by reading `/dev/mem`, any memory in Linux kernel can be read. Copker should be compiled without loadable kernel module (LKM) or KMEM support, to mitigate 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, unprivileged or privileged attackers could not influence Copker's caches. The last condition is satisfied, as Linux enforces an un-swappable kernel space memory.

When Linux crashes, the kernel memory may be dumped to disks automatically. This feature is supported by Kdump, which utilizes `kexec` to quickly boot to a dump-capture kernel. As a result, sensitive data in `cacheCryptoEnv` may be flushed to RAM and contained in the dump. An attacker might exploit this feature to crash the kernel, by inducing system errors. So `kexec` should not be compiled with Copker.

If ACPI state S3 (suspend-to-RAM) or S4 (suspend-to-disk) happens while Copker is in the active mode, we shall ensure that sensitive data are not flushed out. Before ACPI calls (`.prepare` and `.enter`) are issued, 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 will be written to RAM or disks.

6.3 Attacks on Hardware

Attackers might reboot the computer with a malicious booting device (e.g. an external USB drive), attempting to dump the cache content in a way similar to cold-boot attacks. If the cache lines were not cleared after rebooting, the cache content might be captured. However, such attack does not work, since internal caches are invalid after power-up or reset [27]. Even if data might remain in caches (depending on CPU hardware features), read instructions fetch data from RAM, thereby data in caches are overwritten.

DMA attacks [46], [8] are launched from peripherals and bypass the OS security mechanisms. Copker is not designed to withstand such attacks. Fortunately, it can be countered by monitoring bus activities [45] or configuring IOMMU [46]. Finally, the JTAG interface is used by hardware engineers to debug chips. The CPU state can be extracted using the JTAG interface. However, commercial CPUs rarely export JTAG ports.

7 Related Work

Keeping cryptographic keys safe in computer systems is a great challenge. CPU-bound solutions improve full disk encryption by storing AES keys in CPU registers [35], [36], [44], to counter cold-boot attacks [23] and DMA attacks [46]. These systems defeat cold-boot attacks effectively, but are vulnerable to the advanced DMA attack [8] that actively reads and writes values to memory on running computers. Trusted platform modules (TPMs) are dedicated coprocessors for cryptographic computing [52], and these chips may store the master key of Copker, to mitigate the vulnerable window of initial key derivation.

To protect private keys against memory disclosure attacks, K. Harrison and S. Xu suggested one copy of keys in memory [25], and x86 SSE XMM registers are used to store a 1024-bit RSA private key [40]. PRIME [18] implemented 2048-bit RSA on Intel AVX registers against cold-boot attacks, where some non-sensitive intermediate values are stored in RAM. The one-copy principle is strengthened in Copker and PRIME: only one copy of keys during the computations; otherwise, private keys are encrypted in memory. By using Intel AES-NI instructions, the AES implementation of TRESOR [36] is free of timing side channels. Instructions `rdrand` or `rdseed` return hardware random bits in newer Intel CPUs. These extensions are unavailable in

Intel Q8200. It is easy to integrate these extensions into Copker if available. White-box cryptography [11], [12] tries to hide fixed secret keys in binaries, when the binaries are publicly available. However, this approach does not work effectively for asymmetric algorithms.

The cache-as-RAM (CAR) mechanism [31] is adopted in most BIOSes, to support stack before RAM chips are initialized. Employing the CAR method, CARMA [54] built a trusted computing base with a minimal set of hardware components. Copker integrates CAR and TRESOR: caches are used as RAM for cryptographic computations, and an AES key-encryption key is protected by TRESOR. The execution environment of Copker is more complex than BIOSes and it provides services not only for trusted kernel tasks as TRESOR, but also for untrusted user-mode tasks. FrozenCache [39] is the first attempt to employ CAR to mitigate the threat of cold-boot attacks. FrozenCache uses caches as "pure" storage, while Copker uses caches as memory to perform protected private-key computations.

All execution paths are loaded into GPU instruction caches while keys are stored in GPU registers, so the integrity of PixelVault cryptographic service binaries is kept against any malicious process on CPUs [53]. Mimosa [22] employs hardware transactional memory (HTM) to prevent unauthorized access to sensitive data during cryptographic computations. The transactional execution of cryptographic computations aborts and all sensitive data are cleared with HTM, once any other task attempts to access the sensitive data. Sentry encrypts all data of certain applications, and decrypts them in caches [13]. Sentry employs ARM cache features to lock the plain-text data in caches, against cold-boot and DMA attacks.

CPU security features such as Intel TXT, Intel SGX and ARM TrustZone, are employed to build computing environments against physical attacks and/or software attacks [43], [6], [32], [56], [47], [3], [33]. Copker is built on top of CPU cache-fill modes (not designed for security), against cold-boot attacks.

8 Conclusion

We present Copker, a cryptographic engine that computes with private keys without using RAM. During the computations, Copker uses CPU caches as RAM to store all private keys and sensitive states, and ensures that sensitive data never appear in RAM. Therefore, it is secure against physical attacks on the main memory, such as cold-boot attacks. We implement Cokper, and finish a method to verify that the sensitive data are kept in caches only. The prototype supports two typical public-key algorithms, RSA and ECDSA, and also AES-CTR DRBGs. Experiments show that Copker achieves its security goals with good performance and acceptable impact on other programs.

Copker demonstrates a general framework to implement cryptographic algorithms against cold-boot attacks. Various basic cryptographic operations are performed in caches with sensitive data: AES encryption/decryption, long integer computations, precomputation table lookup, and random bit generations. The large size of caches supports stronger keys and more complicated algorithms. Moreover, Copker allows the algorithms to be implemented with high-level programming languages, also making it easy to be extended for other cryptographic algorithms.

Acknowledgments

J. Lin, L. Guan, Z. Ma, L. Xia and J. Jing were partially supported by 973 Program of China (No. 2013CB338001) and Strategic Pilot Project of Chinese Academy of Sciences (No. XDA06010702). B. Luo was partially supported by NSF OIA-1028098, NSF CNS-1422206 and NSF IIS-1513324.

References

[1] ARM, "ARM946E-S technical reference manual," 2007.
[2] C. Arnaud and P.-A. Fouque, "Timing attack against protected RSA-CRT implementation used in PolarSSL," in *Topics in Cryptology - CT-RSA*, 2013, pp. 18–33.
[3] A. Azab, P. Ning *et al.*, "Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world," in *21st ACM Conference on Computer and Communications Security (CCS)*, 2014, pp. 90–102.
[4] E. Barker, W. Barker *et al.*, "SP 800-57: Recommendation for key management - part 1: General (revised)," National Institute of Standards and Technology, Tech. Rep., 2006.
[5] E. Barker and J. Kelsey, "SP 800-90A: Recommendation for random number generation using deterministic random bit generators," National Institute of Standards and Technology, Tech. Rep., 2012.
[6] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 267–283.
[7] D. Bernstein, "Cache-timing attacks on AES," The University of Illinois at Chicago, Tech. Rep., 2005.
[8] E.-O. Blass and W. Robertson, "TRESOR-HUNT: Attacking CPU-bound encryption," in *28th Annual Computer Security Applications Conference (ACSAC)*, 2012, pp. 71–78.
[9] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *8th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2006, pp. 201–215.
[10] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.
[11] S. Chow, P. Eisen *et al.*, "White-box cryptography and an AES implementation," in *9th International Workshop on Selected Areas in Cryptography (SAC)*, 2002, pp. 250–270.
[12] ——, "A white-box DES implementation for DRM applications," in *2nd ACM Workshop on Digital Rights Management (DRM)*, 2002, pp. 1–15.
[13] P. Colp, J. Zhang *et al.*, "Protecting data on smartphones and tablets from memory attacks," in *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 177–189.
[14] J.-S. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," in *1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 1999, pp. 292–302.
[15] Y. Dodis, D. Pointcheval *et al.*, "Security analysis of pseudo-random number generators with input: /dev/random is not robust," in *20th ACM Conference on Computer and Communications Security*, 2012, pp. 626–642.
[16] L. Dorrendorf, Z. Gutterman, and B. Pinkas, "Cryptanalysis of the random number generator of the Windows operating system," *ACM Transactions on Information and System Security*, vol. 13, no. 1, 2009.

[17] U. Drepper, "What every programmer should know about memory," Red Hat, Tech. Rep., 2007.
[18] B. Garmany and T. Müller, "PRIME: Private RSA Infrastructure for Memory-less Encryption," in *29th Annual Computer Security Applications Conference (ACSAC)*, 2013.
[19] D. Genkin, L. Pachmanov, and I. Pipman, "Stealing keys from PCs by radio: Cheap electromagnetic attacks on windowed exponentiation," in *17th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2015.
[20] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs," in *16th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2014, pp. 242–260.
[21] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology - Crypto*, 2014, pp. 444–461.
[22] L. Guan, J. Lin *et al.*, "Protecting private keys against memory disclosure attacks using hardware transactional memory," in *36th IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 3–19.
[23] J. Halderman, S. Schoen *et al.*, "Lest we remember: Cold boot attacks on encryption keys," in *17th USENIX Security Symposium*, 2008, pp. 45–60.
[24] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.
[25] K. Harrison and S. Xu, "Protecting cryptographic keys from memory disclosure attacks," in *37th IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 137–143.
[26] Intel, "Intel 64 and IA-32 architectures optimization reference manual."
[27] ——, "Intel 64 and IA-32 architectures software developer's manual."
[28] G. Klein, K. Elphinstone *et al.*, "seL4: Formal verification of an OS kernel," in *22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 207–220.
[29] C. Koc, "High-speed RSA implementation," RSA Laboratories, Tech. Rep., 1994.
[30] F. Liu, Y. Yarom *et al.*, "Last-level cache side-channel attacks are practical," in *36th IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 605–622.
[31] Y. Lu, L.-T. Lo *et al.*, "CAR: Using cache as RAM in LinuxBIOS," 2006.
[32] C. Marforio, N. Karapanos *et al.*, "Smartphones as practical and secure location verification tokens for payments," in *21st ISOC Symposium on Network and Distributed System Security (NDSS)*, 2014.
[33] J. McCuney, B. Parnoy *et al.*, "Flicker: An execution infrastructure for TCB minimization," in *3rd European Conference on Computer Systems (EuroSys)*, 2008, pp. 315–328.
[34] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.
[35] T. Müller, A. Dewald, and F. Freiling, "AESSE: A cold-boot resistant implementation of AES," in *3rd European Workshop on System Security (EuroSec)*, 2010, pp. 42–47.
[36] T. Müller, F. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *20th USENIX Security Symposium*, 2011, pp. 17–32.
[37] National Institute of Standards and Technology, "FIPS PUB 186-3: Digital signature standard," 2009.
[38] Y. Oren and A. Shamir, "How not to protect PCs from power analysis," in *Advances in Cryptology - Crypto, Rump Session*, 2006.
[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 (INTRUST)*, 2010, pp. 39–59.
[41] T. Ristenpart, E. Tromer *et al.*, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *16th ACM Conference on Computer and Communications Security (CCS)*, 2009, pp. 199–212.
[42] 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.
[43] F. Schuster, M. Costa *et al.*, "VC3: Trustworthy data analytics in the cloud using SGX," in *36th IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 38–54.
[44] P. Simmons, "Security through Amnesia: A software-based solution to the cold boot attack on disk encryption," in *27th Annual Computer Security Applications Conference (ACSAC)*, 2011, pp. 73–82.
[45] P. Stewin, "A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory," in *16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2013.
[46] P. Stewin and I. Bystrov, "Understanding DMA malware," in *9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2013, pp. 21–41.
[47] H. Sun, K. Sun *et al.*, "TrustICE: Hardware-assisted isolated computing environments on mobile devices," in *45th IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2015, pp. 367–378.
[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 polarSSL Project, https://polarssl.org.
[52] Trusted Computing Group, "TPM main specification," 2011.
[53] G. Vasiliadis, E. Athanasopoulos *et al.*, "PixelVault: Using GPUs for securing cryptographic operations," in *21st ACM Conference on Computer and Communications Security (CCS)*, 2014, pp. 1131–1142.
[54] A. Vasudevan, J. McCune *et al.*, "CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms," in *7th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2012, pp. 48–52.
[55] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium*, 2014, pp. 719–732.
[56] N. Zhang, K. Sun *et al.*, "CaSE: Cache-assisted secure execution on ARM processors," in *37th IEEE Symposium on Security and Privacy (S&P)*, 2016.
[57] Y. Zhang, A. Juels *et al.*, "Cross-VM side channels and their use to extract private keys," in *19th ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 305–316.

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

Le Guan received his PhD degree from Institute of Information Engineering, Chinese Academy of Sciences in 2015. He is now a post-doctoral fellow at College of Information Sciences and Technology, Pennsylvania State University. He is interested in system security.

Ziqiang Ma is a PhD student at Institute of Information Engineering, Chinese Academy of Sciences. He is interested in applied cryptography and system security.

Bo Luo received his PhD degree from Pennsylvania State University in 2008. He is an associate professor with EECS Department, the University of Kansas. He is interested in information retrieval, security and privacy.

Luning Xia received his PhD degrees from Graduate University of Chinese Academy of Sciences in 2008. He is an associate professor at Institute of Information Engineering, Chinese Academy of Sciences. His research interest includes network and system 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 professor at Institute of Information Engineering, Chinese Academy of Sciences. His research interest includes network and system security.