Eliminating Remote Cache Timing Side Channels with Optimized Performance

No Author Given

No Institute Given

Abstract. Cache side channels allow a remote attacker to recover the cryptographic keys, by repeatedly invoking the encryption/decryption functions and measuring the execution time. In this paper, we present Warm+Delay, a protection against remote cache timing side-channel attacks, which conditionally warms the caches by reading constant tables and also conditionally delays the executions. With this protection, the encryption/decryption execution time is always independent of data processed (i.e., keys and plaintexts/ciphertexts), which determines the cache access; so it is effective to eliminate remote cache timing side channels. The Warm+Delay scheme is algorithm-independent, and works without any privileged operations on the system. We apply this scheme to AES, and analyze that it achieves the optimized performance. Experimental results (a) confirm the security, by examining the execution time which does not reflect the data cache access, and (b) validate the optimization, by comparing it with different strategies of warm and delay.

Keywords: AES, cache side channel, optimized performance, block cipher, timing side channel

1 Introduction

In the implementations of a cryptographic algorithm, the cryptographic keys could be leaked through side channels on timing [36, 54, 12, 47, 8, 7, 15, 45, 52, 32], electromagnetic fields [29], power [46, 13], ground electric potential [30] or acoustic emanations [31], even when the algorithm is semantically secure. Among these vulnerabilities, remote timing side channels allow an attacker without any system or physical privilege on the computer, to recover the keys by repeatedly invoking the encryption/decryption functions and measuring the execution time [53, 12, 15, 45, 53, 18, 19].

The remote timing side channels exploiting the time difference of data cache misses and hits, called the *remote cache timing side channels* in this paper, are discovered in many cryptographic implementations of block ciphers [12, 15, 9, 53, 54]. In various implementations of block ciphers, table lookup operations controlled by the keys[1] are the primary time-consuming operations. Because

[1] In fact, the lookup operations are controlled by the key and the plaintext, while the plaintext is known to attackers.

accessing cached data is much faster than those in RAM, so cache misses and hits in these table lookup operations are reflected in the execution time and then the information about keys leaks.

Remote cache side channels widely exists in the software implementations of cryptographic algorithms, and are practical to be exploited. Compared with other side channels on electromagnetic fields, power, ground electric potential and acoustic emanations, remote cache-based side-channel attacks do not require special equipments or extra physical access to the system. Compared with other cache-based side-channel attacks [60, 49, 58, 32], remote attacks do not assume any extra operations or privileges on the target system, except the necessary encryption/decryption invocations.

In this paper, we present Warm+Delay, a defense mechanism to eliminate remote cache timing side channels, with optimized performance. Two elementary operations, *warm* (i.e., to fill cache lines by reading constant tables) and *delay* (i.e., to insert padding instructions after the encryption/decryption operations), have been proposed to prevent cache side-channel attacks [47, 32, 12, 20]. Our scheme combines these two operations as follows, to optimize performance while ensure security.

1. Destroy the relationship between the execution time and cache misses/hits, to ensure security. That is, the measured execution time reflects either the best case (i.e., all lookup tables are cached), or the worst case (i.e., none of tables is in caches).
2. Prefer warm for the best case, to optimize the performance. Warm (i.e., to fill cache lines by reading tables) before executing the encryption/decryption operations, if not all tables are cached. However, cached data may be evicted by system activities and the effect of warm is destroyed; in this case, the time of encryption/decryption may reflect the data cache access, and the inserted padding instructions delay the execution to be the worst case.
3. If delay is performed, warm is also done for the next encryption/decryption operation. So, some useless padding instructions are utilized to perform warm instead. Note that warm also consumes CPU cycles.

The proposed Warm+Delay scheme does not require any privileged operation on the system. The conditions to perform warm and delay, are defined as regular timing. Reading constant tables, padding instructions, and timing are commonly supported in computer systems. The scheme is independent of algorithms and implementations. It does not depend on any special design or feature of block ciphers, and it is applicable to different implementations, except that the lookup tables are accessible in the warm operations.

We apply this scheme to AES, and analyze the situations that it produces the optimized performance. It achieves the optimized performance for the AES-128 implementation with 4.25 KB lookup tables [25], when (a) the probability that some entries of lookup tables are evict from caches by system activities during the executions of encryption/decryption, denoted as P_{evict}, is less than 0.2, and (b) the ratio of the delayed time to the time of reading constant tables from either caches or RAM, denoted as k, is less than 57.4. These conditions hold for

continuous AES executions in commodity systems, and confirmed experimentally by the prototype. In fact, our scheme produces the optimized performance for various AES implementations (the lookup tables in size of 4KB, 4.25KB and 5KB) in different key length (128, 192 and 256 bits), when $P_{evict} < 0.2$ and $k < 42.41$. These conditions also hold in commodity systems.

We implement the Warm+Delay scheme with AES. Experiment results on Linux with one Intel Core i7-2600 CPU and 2GB RAM, demonstrate that, the measured time does not reflect the data cache access. We also validate the optimization in different scenarios, by comparing it with different strategies of warm and delay. To the best of our knowledge, this work is the first protection against cache timing side channels with the analysis of optimized performance.

The remainder of this paper is organized as follows. Section 2 presents the background and related works. Section 3 describes the Warm+Delay scheme and analyzes its security. Section 4 proves that our scheme achieves the optimized performance, which is verified by the evaluation in Section 5. Section 6 contains extended discussions. Section 7 draws the conclusion. Appendix A provides the proof of the optimized performance in details, while Appendix B analyzes various implementations of AES with different key lengths.

2 Background and Related Works

2.1 AES and Block Cipher

AES is the most popular block cipher. Each block is 128-bit, and the key is 128, 192 or 256 in bits. AES encryption (or decryption) consists of a certain round of transformations and the number depends on the key length. Each round transformation except the last one, consists of `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` on the 128-bit state, while the last round only performs `SubBytes`, `ShiftRows` and `AddRoundKey`. These steps except `AddRoundKey`, are usually implemented as table lookup operations [25]. `AddRoundKey` consists of bitwise `XOR` operations on the state. Besides, there is no branch in the execution path.

There are similar implementations of other block ciphers: only table lookup and other basic operations, and no branch in the execution path. For example, 3DES, Blowfish [50], CAST128 [10] used in OpenSSH-7.4p1 [2].

2.2 Cache Timing Side Channel

Caches, a small amount of high-speed memory cells located between CPU cores and RAM, are designed to temporarily store the data recently accessed by CPU cores, avoiding accessing the slow RAM. When the CPU core attempts to access a data block, the operation takes place in caches if the data have been cached (i.e., cache hit); otherwise, the data block is firstly read from RAM into caches (i.e., cache miss) and then this operation is performed in caches.

Cache timing side-channel attacks exploit the fact that accessing cached data is about two orders of magnitude faster than those in RAM, to recover the keys

based on the execution time. Typically, it takes 3 to 4 cycles for a read operation in caches, while an operation in RAM takes about 250 cycles [27]. Two typical remote cache-based side channels for block ciphers are discovered as follows.

The internal collision attack. Each cache line contains multiple lookup table entries, and the internal collision is used to denote accessing the entries in the same cache line. The internal collision attack is applied to DES, 3DES and the first round of AES [53, 54], and extended to the second round and last round of AES [9]. This attack is based on the correlation between the encryption time and the average number of cache hits (or misses). For AES, the input plaintext and the first 128 bits of the key are denoted in bytes as $p_0, p_1, ..., p_{15}$ and $k_0, k_1, ..., k_{15}$ respectively, the inputs of the lookup table i ($0 \leq i \leq 3$) in the first round, are $p_{(i+4j)\%16} \oplus k_{(i+4j)\%16}$ where $0 \leq j \leq 3$. When any two of $p_{(i+4j)\%16} \oplus k_{(i+4j)\%16}$ where $0 \leq j \leq 3$ refer to the entries of the lookup table in the same cache line, a shorter encryption time appears[2]. Then the difference of some key bits can be inferred as the values (i.e., $p_{(i+4j)\%16} \oplus p_{(i+4k)\%16}$, $0 \leq j < k \leq 3$) counted most often, which is used to extract the secret key.

The Bernstein's attack. Bernstein's attack [12] is based on the case that the whole AES execution time is correlated with the time for accessing the lookup tables, which depends on the lookup table index. The attacker collects a large number of encryption times for different plaintexts on the duplicated server whose hardware and software configuration is the same as the victim one, with a known key. For each byte of the key, the attacker obtains the lookup table index ($p_i \oplus k_i$ where $0 \leq i < 16$) corresponding to the maximum AES execution time, from the execution on the duplicate server. Then, the attacker infers the key of the victim server, with the obtained lookup table index and known plaintexts.

2.3 Countermeasure

Although eliminating timing side channels remains difficult [22], different countermeasures have been proposed. Software-based methods are mostly based on two strategies: eliminating cache misses, and/or adding confusion to cache misses. Cache misses may be eliminated by not using caches, avoiding shared caches, or cache warming. On the other hand, adding time delay, adding redundant instructions, order skewing, and table randomization add confusion at cache misses. However, existing methods have some disadvantages: (1) not using caches [48, 32, 35, 51], cache warming [17, 54], masking [14, 47], and adding delay [38, 28, 59, 11] would introduce significant overhead; on the other hand, our scheme combines cache warm and delay with optimized performance; (2) using compact tables [12, 17], order skewing [48] or table randomization [43] require modifications to the implementation, hence, they are less practical; (3) modifying the time accuracy [48, 42] fails to prevent the remote attackers from observing the execution time. (4) avoiding the caches being shared [16] and setting caches no-fill mode [51] needs special privileges on the system.

[2] According to [54], a longer encryption time results for the implementation [1] of AES running on 600-MHz Pentium III CPUs.

Hardware designs are proposed to eliminate timing side channels. A hardware-based mechanism presented in [26], allows cache to be configured dynamically to match the need of a process. The new cache design [55, 37] uses partition-locked caches and random permutation caches to prevent cache interference. [56] reduces cache miss rates by dynamic remapping and longer cache indices.

The works [23, 21, 59] provide automated mechanisms to eliminate cache side channels by new options of compilers and programming languages, which can be adopted by our work. Finally, countermeasures attempt to hide the access patterns from the adversary on the other VMS, against cross-VM cache-based side channels [49, 41]. For example, STEALTHMEM [34] allows each VM to load the sensitive data into the locked cache lines, Düppel [61] cleanses time-shared caches, and StopWatch [40] modifies the timings observed by others.

Timing side channel attacks [36, 57, 19, 18] exists for RSA and ECDSA, which are based on instruction caches. The commeasures include generating the unique execution paths [11, 24, 16] and preventing shared resources [34, 16].

3 Eliminating Remote Cache Timing Side Channels

This section presents the threat mode and design goals. Then, the defense scheme is described, and we explain that it effectively eliminate remote cache timing side channels.

3.1 Threat Model and Design Goals

We focus on the remote cache side-channel attacks on block ciphers. The algorithms of block ciphers are publicly known, but the keys are kept secret before the attacks are launched. The implementation details such as source code, operating systems, cache hierarchy and other hardware configurations, are also publicly known; but the running states are unknown to the remote attackers due to non-deterministic interrupts, task scheduling and other system activities. In particular, the implementation is based on lookup tables, and there is no branch in the execution path of encryption/decryption operations. So the execution time depends mostly on cache misses and hits, when accessing lookup tables. The attackers arbitrarily invoke the encryption/decryption functions and measure the time for each invocation. Moreover, the attackers control the invocation, so they can continuously invoke the functions to cache all tables or invoke no function for a long time to let all tables be evicted, with a high probability.

Our scheme attempts to eliminate remote cache timing side channels, with the following properties:

- It is applicable to various block ciphers and transparent to implementations. The scheme is algorithm-independent and works well for any implementation without any modifying the source code of encryption/decryption.
- It requires no privileged operations on the system. All operations introduced by the defense scheme, are available in common computer systems. It works

Algorithm 1 The WARM+DELAY scheme

Input: *key*, *in*
Output: *out*

```
1: function ProtectedCrypt(key, in, out)
2:     start ← GetTime()
3:     Crypt(key, in, out)   // encrypt or decrypt
4:     end ← GetTime()
5:     if end − start > TWOCM then
6:         Warm()
7:         delay ← GetTime()
8:         if delay − start < WET then
9:             Delay(WET − delay + start)
10:        end if
11:    end if
12: end function
```

well either in user space or kernel space, to protect the cryptographic operations in user mode and kernel mode, respectively.
– The parameters are configurable to optimize the performance. Then, it provides optimized performance by configuring appropriate parameters for different algorithms and platforms.

3.2 The Warm+Delay Scheme

Our scheme integrates two strategies to eliminate remote cache timing side channels [47, 32, 12, 20]: *a*) WARM, load the lookup tables into caches before the encryption/decryption operation; and *b*) DELAY, insert padding instructions after the encryption/decryption operation. These two operations cooperatively destroy the relationship between the time measured by attackers and the cache misses/hits during the execution of encryption/decryption. In practice, warm cannot completely eliminate the cache timing side channels by itself, because the effect of cache warm may be countered by system activities, which evict lookup tables from caches. Delay directly controls the time measured by attackers, but it degrades the performance significantly.

The WARM+DELAY scheme is described in Algorithm 1. Given a block cipher, the execution time without cache misses (TWOCM) and the worst execution time (WET) are constant on a certain platform.

Three operations are introduced by this defense scheme: *a*) WARM(), cache warm by accessing lookup tables as constant data, *b*) DELAY(), delay by padding instructions, and *c*) GETTIME(), timing for the conditions of warm and delay. All these operations are algorithms-independent. Moreover, they are implementation-transparent, except that the address of lookup tables is needed in cache warm.

These operations are supported in common computer systems, either in user mode or kernel mode. The conditions of warm and delay are defined by timing. We do not query the status of cache lines to judge whether an entry of the lookup tables is in caches or not, which are unavailable on common platforms.

In this algorithm, there are two constant parameters, WET and TWOCM. WET is defined as the period for one execution of encryption/decryption, in the case that none of lookup tables is cached before the execution. TWOCM is defined as the maximal period for one execution of encryption/decryption, in the case that all lookup tables are in caches before the execution. These two parameters are determined when there is no concurrent interrupts, task scheduling or other system activities.

In general, ProtectedCrypt() are invoked continuously in a loop, to process a great deal of data, and the performance matters in such cases. So Warm() takes effect in the next execution of encryption/decryption. Therefore, if ProtectedCrypt() has been idle for a long time, we suggest an additional invocation of Warm() before ProtectedCrypt(). Note that, even if this "initial" warm is not performed, the security is still ensured by delay and the impact of performance is negligible if the loop of ProtectedCrypt() is long enough.

3.3 Security Analysis

The remote attackers are (assumed to be) able to measure the time of each execution of ProtectedCrypt(). The cache timing side channels come from the relation between the measured time and the data processed; that is, different data (i.e., keys and plaintexts/ciphertexts) determines the lookup table access, resulting in different measured time as some table entries are in caches and others are not. We ensure that the measured time does not leak any information about the status of cache lines, and then destroy the relationship between the execution time and data processed in encryption/decryption.

According to Algorithm 1, the measured time, i.e., the execution time of ProtectedCrypt(), falls into two intervals $(0, TWOCM]$ and $[WET, \infty)$. We prove that, no information about the status of cache lines leaks through the time measured by attackers.

- **Case 1**: The execution time of Crypt() is in $(0, TWOCM]$. It is almost a fixed value for arbitrary data processed, as all tables are cached. When all table entries are in caches, the access time for any entry is constant, resulting in a fixed execution time.[3]
- **Case 2**: The time of encryption/decryption is in $(TWOCM, WET)$. It will be delayed to WET, which is the execution time as all accessed lookup tables are uncached. That is, the encryption/decryption is executed equivalently without any caches. In this case, the cache timing side-channel attackers cannot infer the relation between the measured time and the data processed.
- **Case 3**: The execution time is greater than WET. It results from task scheduling or interrupts. As explained in Section 3.1, the running states such as interrupts, task scheduling and other system activities, are unknown to the remote attackers, so the attackers cannot obtain the actual execution time of encryption/decryption to infer the keys.

[3] The impact of micro-architecture [20] on cache access is discussed in Section 6.2.

In summary, Cases 1 and 2 cannot be exploited in cache timing side-channel attacks, and in Case 3, the actual execution time is obscured by unknown system activities. Moreover, the sequence of Cases 1, 2, and 3, cannot be exploited to construct cache timing side channels. In Case 1, the measured time is almost constant, and no exploitable differential information exists. Cases 2 and 3 happen only if encryption/decryption is interrupted by non-deterministic activities, and the differential information is determined by the system events but not the keys.

In the following, we explain that our scheme successfully prevent typical remote cache timing side channel attacks. To perform the internal collision attacks [15, 9], the attackers need a collection of plaintexts with short (long) measured time due to cache hits (misses). However, protected by the WARM+DELAY scheme, any plaintext may be processed in the short time (i.e., TWOCM) due to cache warm, while any plaintext may be done in the long time (i.e., WET) due to delay. Therefore, the collections depend on the non-deterministic system activities, but not the data processed; or, the collection contains random plaintexts and the attack results will be random.

When the attackers build the time pattern for Bernstein's attack [12], it will be found that, the pattern reflects only the system activities and has nothing to do with the data processed. Since the system activities of target servers are different from those of the duplicated server and unknown to the attackers, this pattern cannot be used to infer the keys.

4 Performance Analysis

4.1 Overview

In this section, we show that our solution is optimal in the category of approaches that do not modify encryption algorithm code. Intuitively, all of the following principles need to be satisfied in order to achieve the optimal performance:

- In the DELAY() operation, the imposed delay, or the execution time with added delay, is the shortest.
- The number of DELAY() operations is minimum.
- In the WARM() operation, only minimum data is loaded into cache.
- The number of WARM() operations is minimum.
- WARM() is more efficient than DELAY(), WARM() is preferred over DELAY().

We examine these principles with our proposed scheme, and derive practical conditions which make it optimal. We first examine the performance of DELAY(), and then analyze the WARM() operation with AES-128 with 4.25KB lookup tables [25]. In the proof, we show that it is statistically the most efficient to load all lookup tables in WARM(), instead of loading parts of the tables (which leads to more DELAY() in the future). Then we identify the best timing for the WARM() operation. Eventually, we derive a practical condition, so that our scheme is optimal. That is, performing warm operation when the execution time lies in $(TWOCM, \infty)$ is the optimal method in our experiment environment. Last, we discuss the impact of different key lengths and different implementations.

4.2 Performance Analysis of Delay

Theorem 1. *In the* DELAY() *operation, delay to WET imposes the minimal overhead while effectively defending against timing side-channel attacks.*

Proof. From the attackers' viewpoint, there are three types of encryption times that are useless (i.e. the execution time is independent of the plaintext input): the shortest time, WET, and any value that is larger than WET. The shortest time means that all the lookup tables are loaded into the cache with no cache miss during encryption. The WET corresponds to the longest execution path, when all lookup tables are not in the cache. When the execution time is longer than WET, the encryption has been disturbed by the OS (e.g. scheduler or interrupt handler), and it is impossible for the attacker to find the real encryption time.

If we delay the encryption time to a random value or a discrete value less than WET, there still exists an observable relationship between the encryption time and the input. The remote attacker can find the relationship by excessively invoking the encryption. Therefore, delaying to WET imposes the shortest delay that is effective against the side-channel attacks. □

Theorem 2. *Performing the* DELAY() *operation when the encryption time is in* $(TWOCM, WET)$ *results the smallest number of* DELAY() *operations.*

Proof. If the AES execution time is less than TWOCM, it means no cache miss has occurred, so that the encryption time is independent of the input. When the observed encryption time is no less than WET, it means that the OS scheduler or interruption handler has been invoked during encryption – the attacker could not recover the actual encryption time to perform timing attacks, because the time for OS scheduler or interruption handler is unpredictable.

When the execution time is in $(TWOCM, WET)$, it is because of cache miss, OS scheduler, or interrupt handler (the overhead is less than WET). The proposed scheme cannot distinguish the actual reason. However, the attacker can distinguish it by repeatedly invoking the encryption using the same input, and further perform timing attacks. Therefore, DELAY() is necessary. □

4.3 Performance Analysis of Warm

We apply the WARM+DELAY scheme to AES-128 implemented as described in Section 2.1. The WARM() operation is applied to the lookup tables, which consume 4.25KB in total. We will show that in this case our WARM+DELAY scheme gets the best performance.

Theorem 3. *For the hardware of commodity computers, loading all the AES lookup tables into the cache is the best warm operation.*

Proof. In AES, WARM() loads five lookup tables into cache. In the case that at least one corresponding cache line is *not* loaded, cache miss may happen, which will in turn trigger DELAY() operation. Therefore, the benefit of not loading N

Table 1: Reduced and introduced time by not loading N cache lines (in cycle).

N	1	2	3	10	20	30	40	50	60	68
$E(T_{delay})$	3575.39	3907.08	3937.85	3941.00	3941.00	3941.00	3941.00	3941.00	3941.00	3941.00
ΔT_{warm}	44.10	88.20	132.30	441.03	882.06	1323.09	1764.12	2205.15	2646.18	2999.00

($N > 0$) cache lines of lookup tables is the saved loading time in WARM(), while the potential cost is the added time due to DELAY(). In the following, we will prove that the expected cost is greater than the benefit, regardless of N.

In AES, the size of each lookup table T_0, T_1, T_2, T_3 is 1KB, while the size of T_4 is 256 Bytes. We assume that the size of a cache line is C Bytes, hence, each of T_0, T_1, T_2, T_3 needs $1024/C$ cache lines, while T_4 needs $256/C$ cache lines. For a cache line utilized by AES look up tables, the probability that it loads T_0, T_1, T_2, T_3 is 16/17, while the probability it loads the content of T_4 is 1/17.

We assume that each round of AES is independent in terms of cache access. Each of T_0, T_1, T_2, T_3 is accessed four times in each of the first nine rounds, and T_4 is accessed 16 times in the last round. Therefore, the probability that one cache line of T_0, T_1, T_2 or T_3 is not accessed in all first 9 rounds is $P_{0-3} = (1-\frac{C}{1024})^{36}$; and the probability of T_4 being not accessed is $P_4 = (1-\frac{C}{256})^{16}$. The expected probability that any cache line is not accessed in one AES execution is $P_e = \frac{16}{17}P_{0-3} + \frac{1}{17}P_4$. Hence, the probability that N cache lines are not accessed in one execution is P_e^N. Therefore, the probability of *invoking* DELAY() *while N cache lines of lookup tables are not in the cache* is shown in Equation 1.

$$P_{delay} = 1 - (\frac{16}{17}P_{0-3} + \frac{1}{17}P_4)^N \tag{1}$$

We assume the execution time of WET is W, and the shortest execution time (TWOCM) is B, then the expected overhead (i.e. the cost) of not loading N cache line of lookup tables (and invoking DELAY()) is

$$E(T_{delay}) = P_{delay} * (W - B) = (1 - (\frac{16}{17}P_{0-3} + \frac{1}{17}P_4)^N)(W - B) \tag{2}$$

The average time saved by not loading one cache line from RAM is denoted as t_{nc}, then the benefit of not loading N cache lines is $\Delta T_{warm} = N * t_{nc}$.

Therefore, we have: (1) if there exists N that $\Delta T_{warm} > E(T_{delay})$, not warming N cache lines provides better performance. (2) If $\forall N > 0, \Delta T_{warm} < E(T_{delay})$, warming all cache lines is always better. (3) If $\exists N > 0, \Delta T_{warm} = E(T_{delay})$, warming or not warming N cache lines are equivalent.

In our experiment hardware, the size of a cache line C is 64 bytes. The time W (i.e. WET) is 4247 CPU cycles, while the shortest AES execution time B is 306 cycles. The average time t_{nc} for loading one cache line of lookup table is 44.10. From Table 1, we can see that the cost of not loading N ($N > 0$) cache lines of lookup tables is much higher than the benefit. Therefore, loading all the lookup tables into the cache provides the best performance in WARM(). □

In our WARM+DELAY scheme, we perform WARM()after the AES execution. If the execution time is less than the WET, we perform WARM() and then

DELAY(). Therefore, the time of WARM() becomes part of delay. In this case, the overhead of WARM()is further reduced. When this case occurs, the extra time reduced by not warming some cache line is zero, if the AES execution time plus WARM() time is less than WET. Otherwise, the extra overhead (e.g. overhead beyond WET) is still less than WARM(). In this situation, warming all cache lines is still the better choice.

Note, in commodity hardware, the cache is enough to load all lookup tables. For example, loading all AES lookup tables needs 5KB, while loading all tables for 3DES needs 2KB. However, the typical L1D cache size is 32 or 64KB for Intel CPU, and 16KB or 32KB for ARM CPU.

Theorem 4. *For commodity computers, performing* WARM() *when cache-miss occurred during the previous execution results the minimum extra time.*

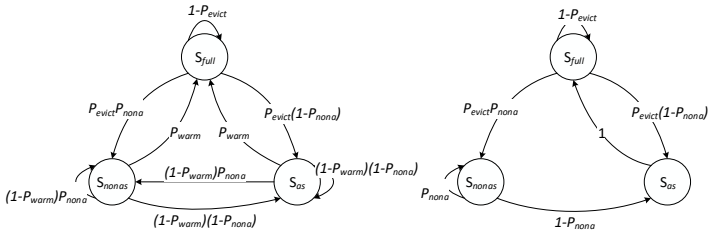
Proof. To prove Theorem 4, we compare it with two other cache warm strategies:

- **Less** WARM() **operations**. When there is a cache miss in the previous AES execution, we perform WARM() with a probability less than 1. In this case, DELAY()is needed. If the next encryption accesses any entry that is currently not in caches, performing WARM() operation before hand introduces none extra overhead – without WARM() all these entries will still be loaded into caches during encryption. If the next encryption accesses some uncached entries, as proved in Thorem 3, when N cache line size of lookup tables are not cached, not performing WARM() introduces more extra time.
- **More** WARM() **operations**. Performing the WARM() operation with a probability P_{warm}, regardless of the cache state after the previous execution. Denote this strategy as *probabilistic* WARM(), while our scheme as *conditional* WARM().

To prove the second case, we categorize the system state into three cases: (1) all the lookup tables are in the cache (S_{full}), (2) the lookup entries not in the cache are not accessed (S_{nonas}), and (3) the lookup entries not in the cache are accessed (S_{as}). We use the Markov model to analyse both *probabilistic* WARM() and *conditional* WARM(). S_{nonas} and S_{as} indicate the states that one or more cache lines of lookup tables are evicted. Figure 1 shown the state transition diagram. In the diagram, P_{nona} (equals to $1 - P_{delay}$) is the probability that the entries not in the cache are not needed in one execution; while P_{evict} is the probability that some entries are evicted from the cache. We use Π^p_{full} (Π^c_{full}), Π^p_{nonas} (Π^c_{nonas}), Π^p_{as} (Π^c_{as}) to represent the limit distribution of the three states when the Markov chain in stable state for the probabilistic and conditional WARM() respectively, and the values are as follows.

$$\Pi^p_{full} = \frac{P_{warm}}{P_{evict} + P_{warm}} \, , \Pi^p_{nonas} = \frac{P_{nona}P_{evict}}{P_{evict} + P_{warm}} \, , \Pi^p_{as} = \frac{P_{evict}(1 - P_{nona})}{P_{evict} + P_{warm}} \tag{3}$$

$$\Pi^c_{full} = \frac{1 - P_{nona}}{1 - P_{nona} + P_{evict}} \, , \Pi^c_{nonas} = \frac{P_{nona}P_{evict}}{1 - P_{nona} + P_{evict}} \, , \Pi^c_{as} = \frac{P_{evict}(1 - P_{nona})}{1 - P_{nona} + P_{evict}} \tag{4}$$



(a) probabilistic warmup (b) conditional warmup

Fig. 1: The Markov state-transferring probability diagram.

Table 2: The values of k_0 corresponding to P_{evict}.

P_{evict}	0	0.05	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
k_0	∞	216.7	109.8	57.4	40.7	33.1	29.0	26.7	25.4	24.8	24.6	24.7

The expected extra time introduced by the probabilistic and conditional WARM() are denoted as $E(T^p)$ and $E(T^c)$, respectively. t_{delay} and t_{warm} are the time needed to perform *one* DELAY() and *one* WARM() operation[4]. Then,

$$E(T^p) - E(T^c) = \Pi_{as}^p * t_{delay} + (1 - \Pi_{as}^p) * P_{warm} * t_{warm} - \Pi_{as}^c * t_{delay} \quad (5)$$

We use $k = t_{delay}/t_{warm}$, so $E(T^p) > E(T^c)$ holds when $1 \leq k \leq k_0$.

$$k_0 = \frac{(P_{delay} + P_{evict}) * (1 + P_{evict} - P_{evict} * P_{delay})}{P_{evict} * P_{delay} * (1 - P_{delay})} \quad (6)$$

From Equation 1, we get $P_{delay} > 0.907$. When P_{evict} varies, the range of k_0 is listed in Table 2 ($P_{delay} = 0.907$). When P_{delay} is larger, k_0 gets larger. For example, in the environment described in Section 4, the maximum of t_{delay} is 3747 cycles, the minimum of t_{warm} is 73 cycles (accessing the lookup tables from caches). t_{warm} is 208 when loading one cache line of the lookup table from RAM, and the maximum of t_{warm} is 2999 when all the lookup tables in RAM. Therefore, the range of k is: $1 < k < 51.33$. When at least one cache line of lookup tables is loaded from RAM, $1 < k < 18.01$. Also, we compute 2^{20} AES encryptions with our scheme, and find that the maximum of P_{evict} is less than 0.02. Therefore, the conditional WARM() provides better performance than the probabilistic WARM(). □

One AES execution is a partial warm operation. When the system is in the state S_{as}, one AES execution is a partial warm operation, as in this case, cache misses are resulted to load the corresponding entries into the cache.

When the system state is S_{as}, our scheme performs WARM() with probability 1, while the probabilistic WARM() performs WARM() with the probability

[4] t_{delay} varies for different executions and t_{warm} varies when loading different size of lookup tables.

Table 3: The introduced time by performing warmup operation or not (in cycle).

N	1	2	3	10	20	30	40	50	60	68
$E(T_{delay})$	3575.39	3907.08	3937.85	3941.00	3941.00	3941.00	3941.00	3941.00	3941.00	3941.00
$E(T_{warm})$	116.03	156.06	202.09	503.29	933.58	1363.87	1794.16	2224.45	2654.74	2999.00

P_{warm}, which is increased by the AES execution, a partial warm operation. However, according to Theorem 4, $E(T^p) > E(T^c)$ holds when $1 \leq k \leq k_0$, which is independent of P_{warm}. Therefore, our scheme is still better.

Process scheduling and interruption occur during the AES execution. In this case, $end - start > WET$, WARM() will be triggered in our scheme. As the process scheduling and interruption occur, we assume that N cache lines of lookup tables are evicted from caches. Hence, the extra overhead introduced by using WARM() to load all lookup tables is $E(T_{warm,N}) = N * t_{nc} + (68 - N) * t_c$, where t_c and t_{nc} denote the time for accessing one cache line from caches and RAM, respectively. If we do not perform WARM(), the next round of AES may need to access part(s) of lookup tables from RAM instead of caches, which will trigger DELAY(). The expected overhead, denoted as $E(T_{delay})$, is shown in Equation 2. Table 3 shows that performing WARM()achieves better performance.

Periodical schedule function is called without scheduling during AES execution. As no other process invoked, the lookup tables are not evicted. Therefore, WARM() is unnecessary although $TWOCM < end - start \leq WET$. However, WARM() is a better choice, as we cannot predict whether the lookup tables are in the cache, and the overhead introduced by accessing the elements in caches is concealed by DELAY().

The relationship between the cache state and execution time. In Theorems 3 and 4, we prove that performing WARM() when not all lookup tables are in caches, results the smallest extra time. However, in our scheme, we perform the WARM() operation according to the previous AES execution time, as we are technically unable to observe the cache state without introducing additional overhead. The relation between the cache state and execution time is as follows:

- $end - start \leq TWOCM$, the system is in state S_{full} or S_{nonas}, no WARM() is needed according to Theorem 4.
- $end - start > WET$, the system is in state S_{as} or S_{nonas}, WARM() is needed according to Theorem 3.
- $TWOCM < end - start \leq WET$, the system is in the state S_{as} or S_{full}, WARM() is better according to previous analysis.

4.4 Other Key Lengths and Other Implementations

The WARM+DELAY scheme provides the optimized performance for various implementations of AES [3, 4] with different key length, once $1 \leq k \leq k_0$. All the proofs above stands, with the only difference be the value of P_{delay} (detailed in Appendix B). P_{delay} depends on the size of lookup tables and the number of iterated rounds. For mbed TLS-1.3.10 [4], OpenSSL-0.9.7i [3] and OpenSSL-1.0.2c [3], the size of lookup tables are 4.25KB, 5KB and 4KB, respectively, and

Table 4: the minimum value of the P_{delay} in different cases.

table size	AES-128	AES-192	AES-256
4.25KB	0.907	0.944	0.966
5KB	0.85	0.88	0.90
4KB	0.924	0.954	0.973

the number of rounds is 10, 12 and 14 for AES-128, AES-192 and AES-256, Table 4 lists the corresponding minimum P_{delay}.

For other table-lookup block ciphers, we have to determine the P_{delay} according to the algorithm and the implementation. Once $1 \leq k \leq k_0$ is satisfied , WARM+DELAY provides the optimized performance.

5 Implementation and Performance Evaluation

The evaluation platform is a Lenovo ThinkCentre M8400t PC with an Intel Core i7-2600 CPU and 2GB RAM. This CPU has 4 cores and each core has a 32KB L1 data cache. The operating system is 32-bit Linux with kernel version 3.6.2.

5.1 Implementation

We apply the WARM+DELAY scheme to AES-128. The implementation of AES employs the mbed TLS-1.3.10 [4]. As we aim to provide the optimized performance while eliminating remote cache timing side channels, efficient GETTIME(), WARM() and DELAY() are finished; and the constant parameters (TWOCM and WET) are determined properly. Next, we show the implementation details about the WARM+DELAY scheme.

TWOCM. TWOCM is larger than the minimum AES execution time (no cache miss occurs), which avoids the unnecessary WARM() and DELAY() operations; and less than the AES execution that only one cache miss occurs. The average minimum AES execution time is measured by average 2^{30} AES execution time with the lookup tables all in caches. In our environment it is 331 cycles. Figure 2 shows the distribution of AES execution time for 2^{30} plaintexts while all lookup

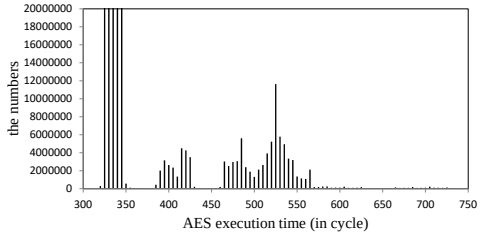


Fig. 2: The distribution of AES execution time only not warm one cache line.

tables except one block of 64 bytes (i.e., one cache line) are cached. Note that, this uncached entry may be unnecessary in an execution of AES encryption. In Figure 2, the data between 380 and 430 cycles are caused by the impact of micro-architecture (see Section 6.2 for details), and the data between 460 and 570 are caused by the cache miss. Finally, we choose 380 cycles as TWOCM.

WET. We flush all cached data before the AES execution and record the AES encryption time. The average value is 4247 CPU cycles.

Warm(). It accesses all blocks of the five lookup tables to load them into caches and every time access one byte of each block of 64 bytes (i.e., the size of one cache line). In order to prevent the compiler optimization, the variables in the function are declared with keyword `volatile`.

GetTime(). We adopt the instruction `RDTSCP` to implement GetTime(), to obtain the current time in high precision (clock cycles) with low cost. `RDTSCP` is a serializing call which prevents the CPU from reordering it. In the implementation, we need to perform the following operations to achieve the high accuracy: (1) as the TSCs on each core are not guaranteed to be synchronized, we install the patch [x86: unify/rewrite SMP TSC sync code] to synchronize the TSCs; (2) the clock cycle changes due to the energy-saving option of the computer, we disable this option in BIOS to ensure the clock cycle be a constant.

Listing 1.1: The implementation of Delay().

```
volatile int delay(uint64_t t_delay){
    uint64_t n = (double)t_delay > 12.886 ?
            (uint64_t)((double)t_delay/2.995-4.302) : 0;
    for (; n>0; n--)
        asm volatile ("xor %%eax, %%eax;" : : : "%eax");
}
```

Delay(). It pads instructions to WET, without making lookup tables evicted from caches. The `usleep()` and `nanosleep()` are inappropriate, as they switch the state of AES execution process to TASK_INTERRUPTIBLE, which may make the lookup tables evicted from caches. We adopt the `xor` instruction to operate data in registers repeatedly, achieving a high precision (in clock cycles) without modifying the cache state. We measure the time cost for different loop number of the `xor` instruction, by invoking it 10^6 times with different loop numbers (from 0 to 2000 and the step is 50). The relation between the time delayed and the loop number of `xor` instruction is calculated through the least squares method. The equation is $t_{delay} = 2.995n + 12.886$, and the coefficient of determination is 0.999993. Implementation of Delay() is provided in Listing 1.1.

Besides, the cost of `RDTSCP` is 36 cycles which is larger than the comparison operation. To achieve better performance, we perform GetTime() only when $delay - start < WET$, instead of every time after Warm(). In this way, if the input of Delay() is less than zero, it simply returns.

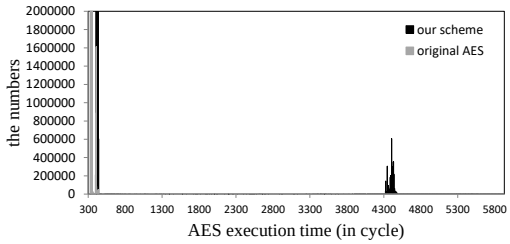


Fig. 3: The observed AES execution time with different plaintext.

5.2 Performance Evaluation

Figure 3 shows the distributions of AES execution time for 2^{30} different plaintexts. It is clearly that most of the execution time is less than TWOCM or no less than WET using the WARM+DELAY scheme. The execution time of our scheme is less than 1.29 times of the unprotected AES.

We evaluate the probabilistic WARM() with the probability 0, 1/2, 1/3 and 1, and our scheme under low and high computing and memory workload when the interval of OS scheduler is 1ms and 4ms respectively. In each case, we perform 2^{30} AES encryptions for random plaintexts. The AES encryption process and the concurrent workload run on the same CPU core. We use the benchmark SysBench to simulate computing and memory reading workload. For computing workload, we run SysBench in its CPU mode, which launches 16 threads to issue 10K requests to search the prime up to 300K. For memory workload, we adopt SysBench with 16 threads in its memory mode, which reads or writes 32KB block each time to operate the total 3GB data on one CPU core.

Figure 4 validates that the WARM+DELAY scheme is the best. Moreover, we calculated P_{evict} under different workloads, according to the number of AES encryption whose execution time is greater than TWOCM. From Table 5, we find P_{evict} is less than 0.005 always, in which case the WARM+DELAY scheme is the optimal as proved in Section 4.

6 Discussions

6.1 Effect of Instruction Caches

Firstly, the implementation of block ciphers is not subject to timing side-channel attacks based on instruction caches [5], because there is generally no branch in

Table 5: The value of P_{evict} under different workload.

Interval of OS scheduler	Low workload	High CPU workload	High mem workload
1ms	0.0028	0.0037	0.0041
4ms	0.0020	0.0026	0.0038

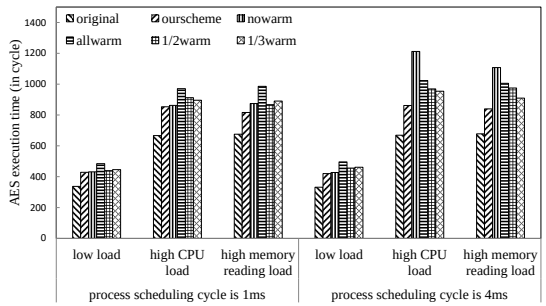


Fig. 4: AES execution performance in different scenarios.

the execution path. The status of instruction caches affect the execution time, but is not related to the data (i.e., keys and plaintexts/ciphertexts). The instructions of block ciphers may be evicted from the caches due to system activities, which causes instruction cache misses and increases the execution time.

As the effect of instruction caches on the execution time is indistinguishable from that of data caches, we determine TWOCM when all instructions of encryption/decryption are cached; and the value will be greater if it is done when the instructions are uncached. However, when the encryption/decryption functions are invoked and all the instructions are cached, such a greater TWOCM will offer attack opportunities because the measured time may leak some information about data cache access. Similarly, we determine WET as the execution time when all instructions are not in instruction caches (and all lookup tables are not in data caches). Otherwise, the measured time may also leak some information about data cache access.

6.2 The Attacks on Fully-Warmed Caches

Even when all lookup tables are in caches, the distribution of encryption time for different plaintext varies slightly, which may be exploited to launch a loaded initial state attack [20]. In our environment, for each of 2^{30} random plaintexts, we perform the encryption after loading all the lookup tables into cache. Figure 5 describes the distribution of encrypting time. The loaded initial state attack [20] predicts part of the key by analysing the plaintext, whose observed encryption time is between 382 and 440 cycles.

When Warm+Delay scheme is adopted, the observed encryption time is either in the interval $(0, TWOCM]$ or $[WET, \infty)$. The loaded state attack cannot infer any key bit directly, when $TWOCM = 381$.

However, for different plaintexts, the probability that the observed time of one encryption falls into $[WET, \infty)$, varies from $5.32 * 10^{-6}$ to $8.13 * 10^{-5}$ (The probability interval is obtained by performing 10^5 encryption for each of 2^{20} random plaintext.). The adversary may attempt to infer the plaintexts for the loaded initial state attack by encrypting the plaintext multiple times. The War-

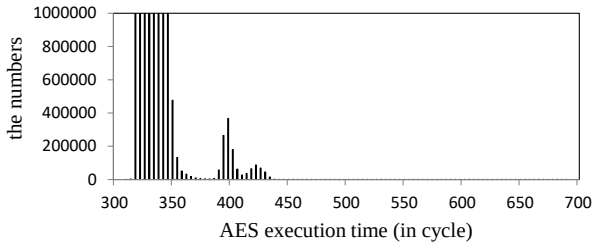


Fig. 5: The distribution of AES encryption time in full warm condition.

M+DELAY scheme may choose a smaller TWOCM, to make the difference of probability negligible, with the performance degradation.

6.3 Other Cache-based Side Channels

Beside the timing attack, there are two other kinds of cache-based side-channel attacks: the trace-driven attack and the access-driven attack. Firstly, both of these attacks require special privileges on the target system. During the execution of encryption/decryption, the trace-driven attackers monitor the variations of electromagnetic fields or power to capture the profile of cache activities and deduce cache hits and misses [25, 39, 7, 6], while the access-driven attackers run a spy process on the target server accessing shared caches, to infer the cache status periodically [44, 47, 32, 33]. The attackers with such privileges are not considered in our scheme, and will be considered in our future work.

In principle, delay is not effective for trace-driven and access-driven attacks, because cache misses and hits exist during the execution of encryption/decryption. On the other hand, warm is effective for these attacks, provided that the cached lookup tables are not evicted by system activities or the local spy process.

7 Conclusion

We propose the WARM+DELAY scheme to eliminate the remote cache timing side channel attacks, for the block ciphers implemented based on lookup tables. Our scheme is applicable to all regular block ciphers, and transparent to implementations. The scheme works well in common computer systems, without any privileged operation.

We prove that, the WARM+DELAY scheme destroys the relationship between the measured time and cache misses/hits, to ensure security. Then the scheme is applied to AES, and analyze the situations that it produces the optimized performance. Experimental results on the prototype system, confirm the security against remote cache timing side channels, and validate the performance optimization.

References

1. ANSI C Reference Code V2.0 (October 24, 2000). National Institute of Standards and Technology. http://csrc.nist.gov/CryptoToolkit/aes/rijndael/
2. OpenSSH. http://www.openssh.com/
3. OpenSSL:Cryptography and SSL/TLS Toolkit. https://www.openssl.org/
4. SSL Library mbed TLS / PolarSSL. https://tls.mbed.org/
5. Acıiçmez, O.: Yet another microarchitectural attack::exploiting I-cache. In: ACM Workshop on Computer Security Architecture. pp. 11–18 (2007)
6. Acıiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (2006)
7. Acıiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (short paper). In: International Conference on Information and Communications Security. pp. 112–121. Springer (2006)
8. Acıiçmez, O., Schindler, W., Koç, Ç.K.: Improving brumley and boneh timing attack on unprotected ssl implementations. In: Proceedings of the 12th ACM conference on Computer and communications security. pp. 139–146. ACM (2005)
9. Acıiçmez, O., Schindler, W., Koç, Ç.K.: Cache based remote timing attack on the AES. In: Cryptographers - Track at the RSA Conference. pp. 271–286. Springer (2007)
10. Adams, C.: The CAST-128 Encryption Algorithm. RFC Editor (1997)
11. Askarov, A., Zhang, D., Myers, A.C.: Predictive black-box mitigation of timing channels. In: ACM Conference on Computer and Communications Security. pp. 297–307 (2010)
12. Bernstein, D.J.: Cache-timing attacks on AES. Vlsi Design IEEE Computer Society 51(2), 218 – 221 (2005)
13. Bertoni, G., Zaccaria, V., Breveglieri, L., Monchiero, M., Palermo, G.: AES power attack based on induced cache miss and countermeasure. In: International Conference on Information Technology: Coding and Computing. pp. 586–591 Vol. 1 (2005)
14. Bl?mer, J., Guajardo, J., Krummel, V.: Provably Secure Masking of AES (2005)
15. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: Cryptographic Hardware and Embedded Systems - CHES 2006, International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings. pp. 201–215 (2006)
16. Braun, B.A., Jana, S., Dan, B.: Robust and efficient elimination of cache and timing side channels. Computer Science (2015)
17. Brickell, E., Graunke, G., Neve, M., Seifert, J.P.: Software mitigations to hedge AES against cache-based software side channel vulnerabilities. (2006)
18. Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: European Conference on Research in Computer Security. pp. 355–371 (2011)
19. Brumley, D., Boneh, D.: Remote timing attacks are practical. Computer Networks the International Journal of Computer & Telecommunications Networking 48(5), 701–716 (2005)
20. Canteaut, A., Lauradoux, C., Seznec, A.: Understanding cache attacks (2006)
21. Cleemput, J.V., Coppens, B., Sutter, B.D.: Compiler mitigations for time attacks on modern x86 processors. Acm Transactions on Architecture & Code Optimization 8(4), 23 (2012)
22. Cock, D., Qian, G., Murray, T., Heiser, G.: The last mile: An empirical study of some timing channels on sel4. pp. 570–581–570–581 (2014)
23. Coppens, B., Verbauwhede, I., Bosschere, K.D., Sutter, B.D.: Practical mitigations for timing-based side-channel attacks on modern x86 processors. In: Security and Privacy, 2009 IEEE Symposium on. pp. 45–60 (2009)

24. Crane, S., Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Thwarting cache side-channel attacks through dynamic software diversity. In: NDSS Symposium (2015)
25. Daemen, J., Rijmen, V.: The design of Rijndael: AES, the advanced encryption standard. Springer-Verlag (2002)
26. Dan, P.: Partitioned cache architecture as a side-channel defence mechanism. Iacr Cryptology Eprint Archive (2005)
27. Drepper, U.: What every programmer should know about memory. Tech. rep., Red Hat, Inc (2007)
28. Ferdinand, C.: Worst case execution time prediction by static program analysis 156, 125 (2004)
29. Genkin, D., Pachmanov, L., Pipman, I.: Stealing keys from PCs by radio: Cheap electromagnetic attacks on windowed exponentiation. In: 17th International Workshop on Cryptographic Hardware and Embedded Systems (CHES) (2015)
30. Genkin, D., Pipman, I., Tromer, E.: 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). pp. 242–260 (2014)
31. Genkin, D., Shamir, A., Tromer, E.: RSA key extraction via low-bandwidth acoustic cryptanalysis. In: Crypto. pp. 444–461 (2014)
32. Gullasch, D., Bangerter, E., Krenn, S.: Cache games – bringing access-based cache attacks on AES to practice. In: IEEE Symposium on Security and Privacy. pp. 490–505 (2011)
33. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a Minute! A fast, Cross-VM Attack on AES. Springer International Publishing (2014)
34. Kim, T., Peinado, M., Mainar-Ruiz, G.: STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In: Usenix Conference on Security Symposium. pp. 11–11 (2012)
35. K?nighofer, R.: A fast and cache-timing resistant implementation of the aes. In: Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings. pp. 187–202 (2008)
36. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: International Cryptology Conference on Advances in Cryptology. pp. 104–113 (1996)
37. Kong, J., Acıiçmez, O., Seifert, J.P., Zhou, H.: Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: IEEE International Symposium on High PERFORMANCE Computer Architecture. pp. 393–404 (2009)
38. Kopf, B., Durmuth, M.: A provably secure and efficient countermeasure against timing attacks. In: Computer Security Foundations Symposium, CSF. IEEE. pp. 324–335 (2009)
39. Lauradoux, C.: Collision attacks on processors with cache and countermeasures. In: WEWoRC 2005 - Western European Workshop on Research in Cryptology, July 5-7, 2005, Leuven, Belgium. pp. 76–85 (2005)
40. Li, P., Gao, D., Reiter, M.K.: Mitigating access-driven timing channels in clouds using stopwatch. In: International Conference on Dependable Systems and Networks. pp. 1–12 (2013)
41. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: Security and Privacy, 2015 IEEE Symposium on. pp. 605–622. IEEE (2015)

42. Martin, R., Demme, J., Sethumadhavan, S.: Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: International Symposium on Computer Architecture. pp. 118–129 (2012)
43. Mer, J., Krummel, V.: Analysis of countermeasures against access driven cache attacks on AES. Springer Berlin Heidelberg (2007)
44. Neve, M., Seifert, J.P.: Advances on access-driven cache attacks on AES. In: Selected Areas in Cryptography, International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers. pp. 147–162 (2006)
45. Neve, M., Seifert, J.P., Wang, Z.: A refined look at Bernstein's AES side-channel analysis. In: ACM Symposium on Information, Computer and Communications Security. pp. 369–369 (2006)
46. Oren, Y., Shamir, A.: How not to protect PCs from power analysis. In: Crypto - Rump Session (2006)
47. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings. pp. 1–20 (2006)
48. Page, D.: Defending against cache-based side-channel attacks. Information Security Technical Report 8(1), 30–44 (2003)
49. Ristenpart, T., Tromer, E., 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). pp. 199–212 (2009)
50. Schneier, B.: Description of a new variable-length key, 64-bit block cipher (blowfish). In: Fast Software Encryption, Cambridge Security Workshop. pp. 191–204 (1993)
51. Taha, Y.H., Abdulh, S.M., Sadalla, N.A., Elshoush, H.: Cache-timing attack against AES crypto system - countermeasures review (2014)
52. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and countermeasures. Journal of Cryptology 23(1), 37–71 (2010)
53. Tsunoo, Y.: Cryptanalysis of block ciphers implemented on computers with cache. Preproceedings of Isita (2002)
54. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of DES implemented on computers with cache. Proc of Ches Springer Lncs 2779, 62–76 (2003)
55. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. In: International Symposium on Computer Architecture. pp. 494–505 (2007)
56. Wang, Z., Lee, R.B.: A novel cache architecture with enhanced performance and security. In: Ieee/acm International Symposium on Microarchitecture. pp. 83–93 (2008)
57. Yarom, Y., Benger, N.: Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. IACR Cryptology ePrint Archive 2014, 140 (2014)
58. Yarom, Y., Falkner, K.: Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In: 23rd USENIX Security Symposium. pp. 719–732 (2014)
59. Zhang, D., Askarov, A., Myers, A.C.: Predictive mitigation of timing channels in interactive systems. In: ACM Conference on Computer and Communications Security. pp. 563–574 (2011)
60. Zhang, Y., Juels, A., et al.: Cross-VM side channels and their use to extract private keys. In: 19th ACM Conference on Computer and Communications Security (CCS). pp. 305–316 (2012)

61. Zhang, Y., Reiter, M.K.: Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 827–838. ACM (2013)

A Appendix A: The the range of k

In this section, we describe how to calculate the range of k in details. From Table 4, we get $0.8 \leq P_{delay} \leq 1$. The conditions $0 \leq P_{warm} \leq 1$, $0 \leq P_{evict} \leq 1$,and $k > 1$ hold, obviously.

By combing Equation 3, 4 and 5, we get Equation 7 and 8. We need to determine the range of k, which makes $E(T^p) - E(T^c) > 0$ (i.e., $y(P_{warm}) > 0$).

$$\begin{aligned} y(P_{warm}) =& (P_{delay} + P_{evict}) * P_{warm}^2 \\ &+ P_{evict} * (P_{delay} + P_{evict})(1 - P_{delay}) * P_{warm} \\ &- k * P_{delay} * P_{evict} * P_{warm} + k * P_{delay}^2 * P_{evict} \ , \end{aligned} \tag{7}$$

$$E(T^p) - E(T^c) = \frac{y(P_{warm})}{(P_{warm} + P_{evict})(P_{delay} + P_{evict})} \tag{8}$$

The symmetry axis of $y(P_{warm})$ is $P_{warm} = P_a$, where P_a is denoted in Equation 9.

$$P_a = \frac{k * P_{evict}P_{delay} + P_{evict}^2P_{delay} + P_{evict}P_{delay}^2 - P_{evict}^2 - P_{evict}P_{delay}}{2(P_{evict} + P_{delay})} \ . \tag{9}$$

Equation 10 holds when $0.8 \leq P_{delay} \leq 1$, $0 \leq P_{evict} \leq 1$, which means $P_a \geq 0$.

$$\begin{aligned} k > 1 >& \frac{P_{evict}^2 + P_{evict}P_{delay} - P_{evict}P_{delay}^2 - P_{evict}^2P_{delay}}{P_{evict}P_{delay}} \\ =& \frac{P_{evict}}{P_{delay}} - P_{delay} + 1 - P_{evict} \ . \end{aligned} \tag{10}$$

1. If $P_a > 1$ (i.e., Equation 11 holds), the minimum value of $y(P_{warm})$ is $y(1)$. When $k\epsilon(k_1, k_0]$, $y(P_{warm}) \geq 0$ holds.

$$k > \frac{(P_{evict} + P_{delay})(2 + P_{evict} - P_{delay}P_{evict})}{P_{evict}P_{delay}} = k_1 \ . \tag{11}$$

$$k_1 < k \leq \frac{(1 + P_{evict} - P_{evict}P_{delay})(P_{evict} + P_{delay})}{(1 - P_{delay})P_{evict}P_{delay}} = k_0 \ . \tag{12}$$

2. If $0 \leq P_a \leq 1$, (i.e., $1 < k \leq k_1$), the minimum value of $y(P_{warm})$ is $y(P_a)$.

$$\begin{aligned} y(P_a) =& \frac{1}{4(P_{evict} + P_{delay})}\{-P_{evict}^2P_{delay}^2k^2 \\ &+ 2P_{evict}P_{delay}(P_{evict} + P_{delay})(2P_{delay} + P_{evict} - P_{evict}P_{delay})k \\ &- P_{evict}^2(P_{evict} + P_{delay})^2(1 - P_{delay})^2\}. \end{aligned} \tag{13}$$

$y(P_a)$ is a function of k (denoted as $f(k)$). As $-P_{evict}^2 P_{delay}^2 < 0$, the minimum of $f(k)$ is either $f(1)$ or $f(k_1)$, for $1 < k \leq k_1$. When $P_{delay} \in [0.8, 1]$, $f(1)$ and $f(k_2)$ is larger than 0, which means $y(P_a) > 0$

$$\begin{aligned} f(1) =& \frac{1}{4(P_{evict} + P_{delay})}\{-(1 - P_{delay})^2 P_{evict}^4 + 2P_{evict}^3 P_{delay}^2 \\ &+ 4P_{evict} P_{delay}^3 + P_{evict}^2 P_{delay}^2 (4 - P_{delay}^2 - 2P_{evict} P_{delay})\}. \end{aligned} \tag{14}$$

$$f(k_1) = (P_{evict} + P_{delay})(2P_{delay} - 1 + P_{evict} P_{delay}(1 - P_{delay})). \tag{15}$$

From the above analysis, we find that when $1 < k \leq k_0$, $E(T^p) \geq E(T^c)$.

B Appendix B: The P_{delay} in different AES key lengthes and different AES implementations

We assume the size of a cache line is C Byte. P_{delay} represents that the probability of performing delay operation with N cache line size of lookup tables not in the cache. So when the AES implementation uses 4.25KB lookup tables, the P_{delay} for AES-128 AES-192 and AES-256 are as follows respectively:

$$P_{delay}^{128} = 1 - (\frac{16}{17}(1 - \frac{C}{1024})^{36} + \frac{1}{17}(1 - \frac{C}{256})^{16})^N, \tag{16}$$

$$P_{delay}^{192} = 1 - (\frac{16}{17}(1 - \frac{C}{1024})^{44} + \frac{1}{17}(1 - \frac{C}{256})^{16})^N, \tag{17}$$

$$P_{delay}^{256} = 1 - (\frac{16}{17}(1 - \frac{C}{1024})^{52} + \frac{1}{17}(1 - \frac{C}{256})^{16})^N. \tag{18}$$

When the AES implementation uses 5KB lookup tables, the P_{delay} for AES-128 AES-192 and AES-256 are as follows respectively:

$$P_{delay}^{128} = 1 - (\frac{4}{5}(1 - \frac{C}{1024})^{36} + \frac{1}{5}(1 - \frac{C}{1024})^{16})^N, \tag{19}$$

$$P_{delay}^{192} = 1 - (\frac{4}{5}(1 - \frac{C}{1024})^{44} + \frac{1}{5}(1 - \frac{C}{1024})^{16})^N, \tag{20}$$

$$P_{delay}^{256} = 1 - (\frac{4}{5}(1 - \frac{C}{1024})^{52} + \frac{1}{5}(1 - \frac{C}{1024})^{16})^N. \tag{21}$$

When the AES implementation uses 4KB lookup tables, the P_{delay} for AES-128, AES-192 and AES-256 are as follows respectively:

$$P_{delay}^{128} = 1 - ((1 - \frac{C}{1024})^{40})^N, \tag{22}$$

$$P_{delay}^{192} = 1 - ((1 - \frac{C}{1024})^{48})^N, \tag{23}$$

$$P_{delay}^{256} = 1 - ((1 - \frac{C}{1024})^{56})^N. \tag{24}$$