Deconstructing New Cache Designs for Thwarting Software Cache-based Side Channel Attacks

Jingfei Kong
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816, USA
jfkong@cs.ucf.edu

Onur Acıiçmez
and
Jean-Pierre Seifert
Samsung Electronics
95 West Plumeria Drive, San Jose, CA 95134, USA
onur.aciicmez@gmail.com
j.seifert@samsung.com

Huiyang Zhou
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816, USA
zhou@cs.ucf.edu

ABSTRACT

Software cache-based side channel attacks present a serious threat to computer systems. Previously proposed countermeasures were either too costly for practical use or only effective against particular attacks. Thus, a recent work identified cache interferences in general as the root cause and proposed two new cache designs, namely partition-locked cache (PLcache) and random permutation cache (RPcache), to defeat cache-based side channel attacks by eliminating/obfuscating cache interferences. In this paper, we analyze these new cache designs and identify significant vulnerabilities and shortcomings of those new cache designs. We also propose possible solutions and improvements over the original new cache designs to overcome the identified shortcomings.

Categories and Subject Descriptors

E.3 [**Data Encryption**]: [Code breaking]; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Security

Keywords

Cache Architecture, Cryptanalysis, Cache Attack, Timing Attack, Side-Channel Analysis, Microarchitectural Analysis

1. INTRODUCTION

Concrete implementations of theoretically "bullet-proof" cryptographic algorithms may possess certain weaknesses due to physical properties of those implementations. For example, an adversary can observe so-called "side channel'" information such as the power consumption of a cryptographic chip or the execution times of cryptographic applications to derive confidential information, particularly secret keys.

Although previous efforts were often focused on embedded systems like smart cards, practical side channel attacks have been demonstrated on modern commodity computer systems [12, 7, 23]. Recent software-based side channel attacks exploiting architectural features such as data caches ([6, 10, 11, 16, 19, 22]), instruction caches ([2]), shared functional units ([5]) and branch predictors ([3, 4]) are gaining increased attention as these attacks do not require physical device access and only conduct legitimate activities. As a result, they pose serious threats to the security of computer systems. This is also apparent from the fact that widely used cryptographic software, e.g. OpenSSL, have gone through several revisions (e.g. [17, 18]) to strengthen their implementations against side channel cryptanalysis techniques. Furthermore, the recent AES instruction addition to the x86 ISA from Intel explicitly mentions protection against software-based side-channels as one of the justifications for the new AES instructions [15].

Moreover, this microarchitectural threat gets much more severe due to recent developments, which propose to enable the so called performance counter paradigm also within the user mode level (CPL=3) of the Intel x86 or AMD64 architecture, cf. [9]. Here, the goal is to enable modules such as dynamic optimizers and managed runtime environments to monitor the currently running program with very high accuracy and resolution, thereby allowing them to report on performance problems and opportunities and fix them immediately. Although this is nowadays a definitely required feature within the growing Web-based and Java-oriented software engineering efforts, the security implications are also very clear as pointed by AMD. I.e., "...the operating system must ensure that information does not leak from one process to another or from the kernel to a user process", cf. [9].

Although caches are highly effective in reducing average memory access time and thus widely used in modern processors, their internal functionalities, i.e., hit/miss behaviors, were shown to leak critical information that puts trusted software implementations in an unforeseen danger. Of course, some mitigation methods were proposed to defend against software cache-based side channel attacks immediately after the realization of cache architectures as a new side channel source for malicious attacks [20, 10, 19, 22]. However, as also stated in [25], we have seen only what we can call "ad-hoc" solutions to these security problems so far. In other words, each different cache-based side-channel vulnerability was analyzed separately and the software mitigation were proposed in a case-by-case basis. Furthermore, one needs to employ all of these countermeasures together, which brings significant performance overhead, in an implementation to achieve a reasonable security level.

Wang et. al. realized the lack of a comprehensive hardware solution to mitigate cache attacks with low performance overhead and tried to address this issue by proposing new cache designs [25]. They analyzed some cache-based side-channel attacks, identified their root causes, and proposed two different cache architectures, Partition-Locked cache (PLcache) and Random Permutation cache (RPcache). Unfortunately, the authors of [25] did not consider every known cache attack, especially the type of attacks known as cache-collision attacks. Therefore, their proposals do not provide sufficient security levels as will be shown in later sections.

In this paper, we analyze these two cache designs and empirically prove that they fall short on avoiding some cache attacks. We simulated a MIPS R10000 processor with these new cache designs using a timing simulator tool, which is built upon the Simplescalar toolset [13]. We ran several cache attacks on this simulated CPU and used OpenSSL's AES implementation as our target cryptosystem application. Our results show that these cache architectures are still vulnerable to cache attacks. Eventually, we will discuss possible solutions and improvements over the original designs to overcome the identified shortcomings of PLcache and RPcache.

Our paper is organized as follows. The following section gives a short introduction into AES, its usual software implementation and reviews also the known cache-based side-channel attacks against AES. The next section presents then the recently proposed new cache designs from Wang et. al. [25], while section 4 analyzes the respective security of these new cache designs. After having presented in section 4 their security shortcomings, we will discuss in section 5 some potential improvements of the new cache designs from Wang et. al. [25]. The paper finishes with some conclusions in section 6.

2. BACKGROUND

2.1 AES and Its Software Implementation

Rijndael ([14]) is a symmetric block cipher, which was announced as Advanced Encryption Standard (AES) by NIST [8]. AES allows key sizes of 128, 192, and 256 bits and operates on 128-bit blocks. For simplicity, we will describe only the 128-bit version of the algorithm in this paper.

AES performs operations on a 4x4 byte-matrix, called State, which is the basic data structure of the algorithm. The algorithm is composed of a certain number of rounds depending on the length of the key. When the key is 128 bits long, the encryption algorithm has 10 rounds of computations, all except the last one of which performs the same operations. Each round has different component functions and a round key, which is derived from the original cipherkey. The four component functions are

- SubBytes Transformation,
- ShiftRows Transformation,
- MixColumns Transformation,
- and AddRoundKey Operation.

The AES encryption algorithm has an initial application of the AddRoundKey operation followed by 9 rounds and a final round. The first 9 rounds use all of these component functions in the given order. The MixColumns Transformation is excluded in the last round. A separate key scheduling function is used to generate all of the round keys, which are also represented as 4x4 byte-matrices, from the initial key. The details of the algorithm can be found in [14] and [8].

The most widely used software implementation of AES is described in [14] and it is designed especially for 32-bit architectures. To speed up encrytion, all of the component functions of AES, except AddRoundKey, are combined into lookup tables and the rounds turn to be composed of table lookups and bitwise exclusive-or (XOR) operations. The five lookup tables T0, T1, T2, T3, T4 employed in this implementation are generated from the actual AES S-box values as the following way:

$$\begin{aligned}
T0[x] &= (2 \bullet s(x), s(x), s(x), 3 \bullet s(x)), \\
T1[x] &= (3 \bullet s(x), 2 \bullet s(x), s(x), s(x)), \\
T2[x] &= (s(x), 3 \bullet s(x), 2 \bullet s(x), s(x)), \\
T3[x] &= (s(x), s(x), 3 \bullet s(x), 2 \bullet s(x)), \\
T4[x] &= (s(x), s(x), s(x), s(x)) \ ,
\end{aligned}$$

where $s(x)$ and $\bullet$ stand for the result of an AES S-box lookup for the input value x and the finite field multiplication in $GF(2^8)$ as it is realized in AES, respectively.

As shown in Figure 1, in a typical 128-bit-key 10-round AES implementation, each round has two inputs, 16-byte x_0^i, ..., x_{15}^i, which is the output from the previous round, and 16-byte round key k_0^i, ..., k_{15}^i, which are pre-computed from the 16-byte secret key k_0, ..., k_{15}. Each of the lookup tables T0, T1, T2, and T3 contains 256 pre-computed 4-byte values. The initial state x_0^0, ..., x_{15}^0 is computed by 16-byte plaintext p_0, ..., p_{15} XORed with the key k_0, ..., k_{15}. The last round uses another set of tables (T4, T4, T4, T4) and its output is the ciphertext.

Since the initial state, which is computed by XORing the plaintext and the secret key, is used as the table indices in the first round; an adversary can recover the key if the plaintext and the table indices are known . The strength of AES depends on the infeasibility of recovering the table indices, which is a valid argument in the context of theoretical cryptanalysis, even in the case when an adversary has an unlimited number of plaintext and ciphertext pairs computed under the same secret key. Although AES is (still) secure in theory, cache attacks can, in practice, directly exploit the implementation weaknesses to recover the table indices.

2.2 Software Cache-based Side-Channel Attacks

We have seen an increased research efforts on the side-channel analysis of commodity PC platforms for the last few years, especially after the realization of some processor components causing serious side-channel leakages. These efforts led to the development of the new Microarchitectural

$$\begin{Bmatrix} x_0^{i+1}, x_1^{i+1}, x_2^{i+1}, x_3^{i+1} \\ x_4^{i+1}, x_5^{i+1}, x_6^{i+1}, x_7^{i+1} \\ x_8^{i+1}, x_9^{i+1}, x_{10}^{i+1}, x_{11}^{i+1} \\ x_{12}^{i+1}, x_{13}^{i+1}, x_{14}^{i+1}, x_{15}^{i+1} \end{Bmatrix} = \begin{Bmatrix} T_0[x_0^i] \oplus T_1[x_5^i] \oplus T_2[x_{10}^i] \oplus T_3[x_{15}^i] \oplus \{k_0^i, k_1^i, k_2^i, k_3^i\} \\ T_0[x_4^i] \oplus T_1[x_9^i] \oplus T_2[x_{14}^i] \oplus T_3[x_3^i] \oplus \{k_4^i, k_5^i, k_6^i, k_7^i\} \\ T_0[x_8^i] \oplus T_1[x_{13}^i] \oplus T_2[x_2^i] \oplus T_3[x_7^i] \oplus \{k_8^i, k_9^i, k_{10}^i, k_{11}^i\} \\ T_0[x_{12}^i] \oplus T_1[x_1^i] \oplus T_2[x_6^i] \oplus T_3[x_{11}^i] \oplus \{k_{12}^i, k_{13}^i, k_{14}^i, k_{15}^i\} \end{Bmatrix}$$

Figure 1: Round computations in AES.

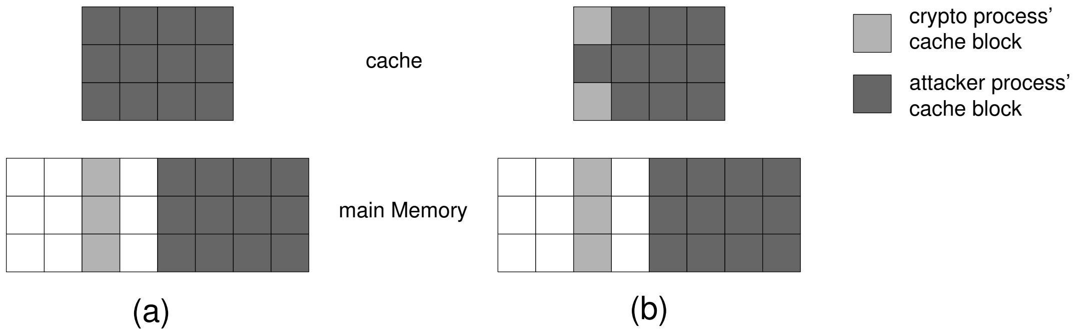


Figure 2: An example of access-driven attacks: prime and probe caches.

Analysis (MA) research area. Those attacks exploit the microarchitectural components of a processor to reveal cryptographic keys. The functionality of some processor components such as cache and branch predictors generates data dependent variations in execution time and power consumption during the execution of cryptosystems. These variations either directly gives the key value out during a single cipher execution (c.f. [3]) or leaks information which can be gathered during many executions and analyzed to compromise the system (c.f. [19, 10, 16]).

There are currently four main types of MA attacks in the literature: Data Cache, Instruction Cache Attacks, Branch Prediction Analysis, and Shared Functional Unit Attacks. In this paper, our focus will only be on data cache attacks, which are usually referred to as cache-based side-channel attacks or simply cache attacks in the literature.

Cache attacks exploit the cache behavior of a cryptosystem by obtaining the execution time and/or the power consumption variations generated via cache hits and misses. Cache analysis techniques enable an unprivileged process to attack another process, e.g., a cipher process, running in parallel on the same processor as done in [19, 16, 22]. Furthermore, some of the cache attacks can even be carried out remotely, e.g., over a local network [6].

Theoretical cache attacks were first described by Page in [20]. He characterized two types of cache attacks: trace-driven and time-driven. Later, we saw another type of cache attack, which is now referred as access-driven, in [22, 19]. We will discuss time-driven and access-driven attacks in this paper and exclude trace-driven attacks due to the fact that there is no instance of a software based trace-driven attack in the literature.

2.2.1 *Access-driven Cache Attacks*

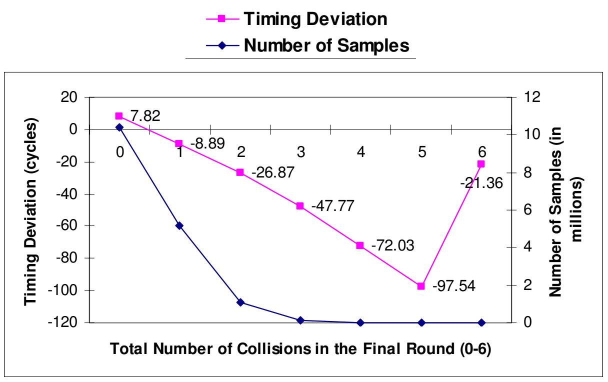
Figure 2 illustrates access-driven attacks on a target crypto application — AES as in our case. A typical access-driven attack consists of the following steps:

1. The attacker occupies the entire cache with his own data (Figure 2a). This can be accomplished by an attack process/thread through loading a large array into the cache.

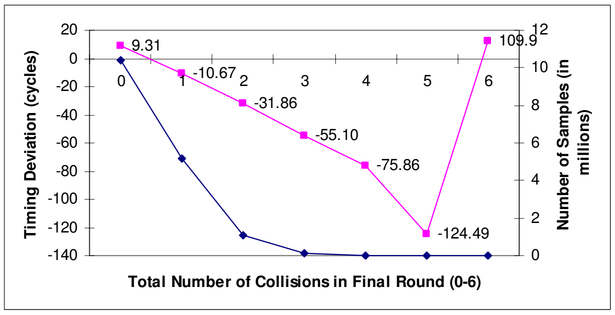
2. He then invokes an AES execution. During execution, the cipher issues table lookups with the indices depending on the key and the plaintext, and the corresponding parts of the AES tables will be loaded into the cache (Figure 2b). Since some parts of the attacker's array will be evicted from the cache, the cache state after the encryption becomes dependent on the accessed table indices and thus leaks information of the secret key.[1]

3. Shortly after the encryption, the attacker reads again its own large array, but this time he also measures the time to read each individual cache line. Since the

[1]The fact that one single cache line contains multiple table elements and multiple rounds share the same set of tables do reduce the leaked information. However, the full key can still be recovered statistically as discussed in [19]. It is also the same case for time-driven attacks.



(a) Pentium 4 Processor



(b) Simulator Processor

Figure 3: The relationship between the number of collisions in the final round and the encryption time.

> cache lines that have been already evicted from the cache require a longer time to read compared to the data still inside the cache, he knows which cache lines were replaced by the AES execution. In other words, the attacker can infer which parts of the AES lookup tables have been accessed during the encryption.

Neve et al. [16] have demonstrated that an adversary can get fine-grained cache behavior snapshots of an AES process on single-threaded processors by manipulating the OS scheduling. It is also shown there that observing a small number of encryptions under the same key provides sufficient information to recover a full 128-bit AES key. Multi-threaded architectures facilitate this class of attacks since attackers can use a hardware-assisted spy process to monitor the execution of the crypto process on-the-fly during the encryption, as demonstrated by Aciiçmez [2], Osvik et. al. [19], and Percival [22]. Although the attacks in [2] and [22] are against RSA, the RSA-vulnerability associated with table lookups is similar to the AES attacks presented in [19]. All these access-driven attacks exploit the same root cause: *the shared cache among processes/threads.*

2.2.2 *Time-driven Cache Attacks*

The first cache-based timing attack against AES was introduced by Bernstein [10]. His attack exploits the statistical correlations between the AES encryption time variations and inputs to the encryption, i.e., plaintext and cipher key. This attack was analyzed in [25] and cache interferences are identified as the root cause. However, Bernstein's attack is not the only time-driven cache attack. A more recent time-driven attack, named "cache-collision attack", has different characteristics. Therefore, the countermeasures designed to defeat Bernstein's attack, including the ones proposed in [25], may not provide sufficient protection against cache-collision attacks, which is the case as shown in this paper.

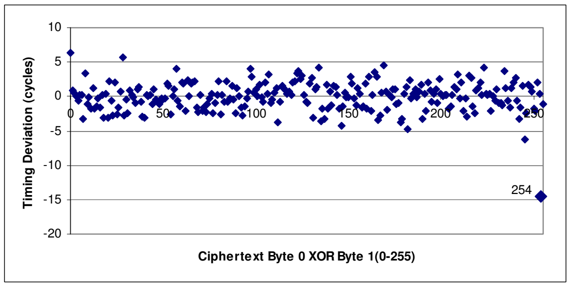
An AES encryption operation can be viewed as a sequence of table lookups and some additional computations. A "cache collision" occurs when two lookups refer to the same element in a table. In this case, the second lookup will certainly be a cache hit. If there is no cache collision, the second lookup may be a cache miss. *Statistically, a higher number of cache collisions lead to a smaller number of cache misses in an encryption and thus a shorter encryption time compared to encryptions with a smaller number (or zero) of cache collisions.* This statement has been experimentally verified in [6, 11, 24]. We have also confirmed it by re-producing the experimental results of the so-called last round AES attack described in [11, 24] on both a real Pentium 4 machine and a simulated processor model (our processor configuration is given in Table 1). Our results [2] are shown in Figure 3 and they are computed from 16 million encryptions (each encryption starts with a clean cache) of 16-byte random plaintexts. Each point in this figure shows the variation between the mean encryption time based on the runs with the same number of collisions and the overall mean encryption time. One can see that the mean encryption time decreases as the total number of collisions for the last round increases. The non-decreasing effects of the last data point (encryption runs with 6 collisions) are mainly due to an insufficient number of the samples with such a high number of cache collisions, thereby not statistically significant.

Cache collision attacks work as follows: Assume there are two table lookups $k_i \oplus x_i$ and $k_j \oplus x_j$ where k_i, k_j are the AES encryption key bytes and x_i, x_j are plaintext bytes for the first round. If used for the last round, k_i, k_j are expanded last round key bytes and x_i, x_j are ciphertext bytes. The essence of the attack is to find the correct key difference between k_i and k_j, i.e., $k_i \oplus k_j$. If there is a collision between $k_i \oplus x_i$ and $k_j \oplus x_j$, the following equation holds:

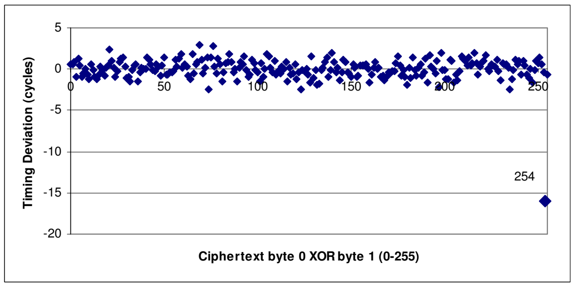
$$k_i \oplus x_i = k_j \oplus x_j \Longleftrightarrow k_i \oplus k_j = x_i \oplus x_j \qquad (1)$$

From a large number of samples, attackers can expect that among all possible 256 values of $x_i \oplus x_j$, the one with the smallest mean encryption time implies a cache collision and therefore corresponds to the correct value of the key byte difference $k_i \oplus k_j$. Figure 4 shows one such example for the last round attack. The figure shows the mean encryption time for each of the 256 values of $x_0 \oplus x_1$, where x_0,x_1 are the first two bytes of the ciphertext. Here, the mean encryption times are computed from 16 million encryptions under the same key on both a real Pentium 4 machine and our simulated processor. The actual difference between the first key byte and the second key byte (i.e. $k_0 \oplus k_1$) is 254.

[2]In this paper, timing deviation is defined as the difference between the mean encryption time among the samples with the same feature, e.g., the same number of collisions, and the mean encryption time among all the samples.



(a) Pentium 4 Processor



(b) Simulator Processor

Figure 4: The relationship between the mean encryption time and $x_0 \oplus x_1$ given $k_0 \oplus k_1$ being 254.

As shown in Figure 4, among all possible values of $x_0 \oplus x_1$, the one with the smallest encryption time correctly reveals the value of $k_0 \oplus k_1$. As a result, an attacker only needs to record the encryption time and the ciphertext to derive the relationship among different key bytes. Then, an attacker can conduct off-line analysis to recover the complete key. Certain searching algorithms can also be used to dramatically reduce the number of samples needed to recover the key. Further details of cache collision attacks can be found in [1, 6, 11, 24].

In summary, current time-driven attacks exploit the fact that access to different levels of the memory hierarchy can introduce observable time differences. If certain aspects of the secret key correlate to the number of cache hits/misses, the cipher is endangered by time-driven attacks. Furthermore, if attackers have the ability to set the initial "clean cache" state, the number of required samples is reduced as it is more likely for cache collisions to lead to reduced encryption time.

3. RECENTLY PROPOSED NEW CACHE DESIGNS FOR THWARTING SOFTWARE CACHE-BASED SIDE CHANNEL ATTACKS

As stated above, we have seen only what we can call "ad-hoc" solutions to the cache side-channel problem so far. In other words, different cache-based side-channel vulnerabilities were analyzed separately and the software countermeasures were proposed in a case-by-case basis as outlined above. One needs to employ many of these countermeasures together, which may incur significant performance overhead, to achieve a reasonable security level. Therefore, we evidently need comprehensive, robust, and efficient defense techniques. Along this direction, Wang et. al. [25] analyzed the Berstein's time-driven attack [10] and Percival's access-driven attack [22]. They identified cache internal interference (interferences from the same process) and external interference (interferences from different processes) as the root cause of cache-based attacks and proposed two new cache designs to overcome the cache interference effect.

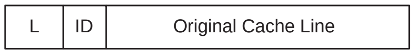


Figure 5: The new cache line of the PLcache

3.1 Partition-Locked Cache

The concept of using partitioned caches against software cache-based side channel attacks was introduced in [21]. It is used to isolate the protected processes from others so that cache interference among different processes (i.e., external interferences) becomes infeasible. Besides incurring too much performance cost, this approach does not solve the internal interference within the same process. The Partition-Locked cache (PLcache) solves those problems with a fine-grain locking over the cache lines. With support from PLcache, software applications are able to lock critical data in the cache in the granularity of one cache line size. Once a cache line is locked in the cache, both external and internal interferences cannot evict it. Therefore, an attacker cannot observe the access pattern of the protected cache lines and can not gain any useful information from the time variance behavior associated with the accesses of those cache lines.

The PLcache design mainly includes two parts: hardware support for locking and architecture exposures for software applications. In the PLcache, each cache line has been augmented with an ID field and a lock bit L. As shown in Figure 5, the ID indicates the owner of the cache line, normally a process. The lock bit L indicates the locking state of the cache line. These two fields will help cache replacement to decide whether a cache line should be replaced as usual or not. For the cache lines of the same process, the basic rule is that incoming non-locked cache lines can't replace locked cache lines. For the cache lines of different processes, the basic rule is that no incoming cache line of one process can replace any locked cache line of another process. For the interface to software applications, special instructions (ld_lock, ld_unlock, st_lock, st_unlock) are introduced. They are used to update the lock bit to control which cache lines should be locked or unlocked. This way, software application can use these special instructions to prevent their critical data being replaced by interferences, thereby defeating some cache attacks.

PLcache has several advantages. The design addresses the root cause of two analyzed attacks - cache interferences and thus provide generic countermeasure support against attacks based on that. Because of fine-grain locking control over cache lines, it was also shown to have low performance overhead on AES. In the meanwhile, however, excessive locking could cause unfairness problems and it was proposed to have the operating system to manage the locking requests from processes.

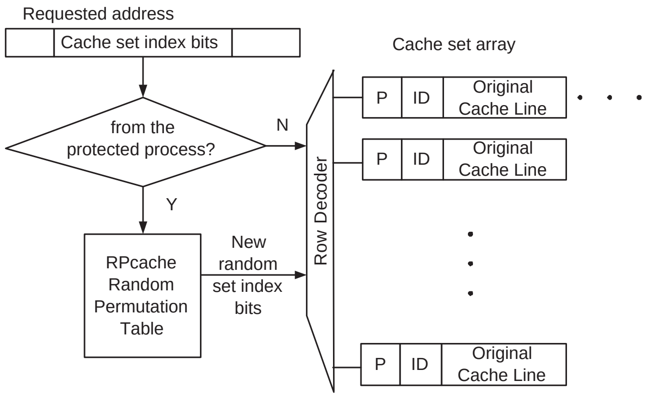


Figure 6: One logical view of the RPcache

3.2 Random-Permutation Cache

Rather than eliminating cache interferences as the PLcache aims to achieve, the Random-Permutation cache (RPcache) takes another perspective. It allows cache sharing and cache interferences but it obfuscates the interferences so that no useful information can be derived. A spy process can observe another process' cache access only if the victim process replaces the spy process' cache lines deterministically. The RPcache improves the security by making such replacements random. In other words, the cache line address in the RP-cache is generated in a random instead of a pre-determined manner.

In the RPcache, each cache line is augmented with one protection bit and an ID. As shown in Figure 6, the protection bit P indicates whether the cache line should be protected. The ID indicates the owner of the cache line, normally a process. In a case of interference, when the fetched cache line and the chosen replacement cache line belong to two different processes, the original cache set will not be used. Instead, another cache set is chosen randomly and replacement happens in that set. This process changes the mapping between addresses and cache sets. To support it, a hardware permutation table is introduced in the RPcache. As shown in Figure 6, typically the protected process will use the hardware random permutation table and thus indirectly address the cache. Other processes will have no permutation tables and therefore no address indirections. It was shown that with such random permutation cache interferences leak no information. Note the P bit is used to defend against the cache attack in [10] and details about the defense can be found in [25].

The novel RPcache design provides the security protection against some analyzed cache attacks, particularly access-driven attacks. In addition, it has the advantage of low performance overhead and requires no changes in the programs.

4. SECURITY ANALYSIS OF THE NEW CACHE DESIGNS

This section discusses the security problems of PLcache and RPcache that we could identify. We have conducted several experiments to support our claims. We implemented both the PLcache and RPcache in our timing simulator. Our simulator is developed upon the Simplescalar toolset [13]. The simulator models MIPS R10000 processor, with the default configuration shown in Table 1.

Superscalar Core	7-stage pipeline: Fetch/Dispatch/Issue/RegisterRead /EXE/WriteBack/Retire
	Fetch/Dispatch/Issue/ MEM issue/Retire Bandwidth: 4
	Fully-symmetric Function Units: 4
	Reorder Buffer size: 64
	Issue Queue Size: 32
	Load Store Queue Size: 32
Execution Latencies	Address Generation: 1 cycle
	Memory Access: 2 cycles (hit in data cache)
	Integer ALU ops: 1 cycle
	Complex ops:MIPS R10000 latencies
Instruction Cache	32KB 2-way, Block size: 64B 10-cycle miss penalty
L1 Data Cache	32KB 2-way, Block size: 64B 10-cycle miss penalty
L2 Unified Cache	2MB 16-way, Block size: 64B 300-cycle miss penalty

Table 1: Processor Configuration

4.1 Analysis of PLcache

The PLcache design has two security deficiencies. The first security vulnerability of the PLcache lies in the initial phases of process execution. Only after all the critical data of a crypto process are loaded into the cache, the PLcache ensures that they are locked and stay in the cache. However, when this crypto process starts to run, the critical data are gradually loaded into the cache, making it vulnerable to either access-driven or cache-collision attacks.

To illustrate this problem, we first ran an access-driven attack on AES using our simulated processor model. In this experiment, we started an AES process with a cold cache and ran a spy process to observe the cache usage of the cipher. Our spy routine asked the AES process to encrypt single 16-byte message blocks and observed cache sets after each encryption. The measurements of this spy process indicate which blocks of AES tables had been accessed and locked in the cache, as shown in Figure 7. For simplicity, we only show the part of the cache that holds only one AES table. The dark boxes represents the cache lines (or blocks of this AES table) locked by the crypto process. As shown from this figure, the PLcache leaks the cache usage information of AES, which enables successful access-driven attacks. It is reported that on average less than 15 clean observations like the one in our experiment are sufficient to completely break a 128-bit AES key [16].

Similarly, for cache-collision attacks, if an adversary can set up a clean cache state before each encryption run, the encryption time correlates to the number of cache collisions. We implemented the PLcache in our timing processor simulator and used the analysis tool from Bonneau's website [3] to conduct last round cache-collision attack. In this attack, each encryption run starts with a clean cache state. The encryption time and the cipher text are collected to derive

[3]http://www.jbonneau.com/research.html

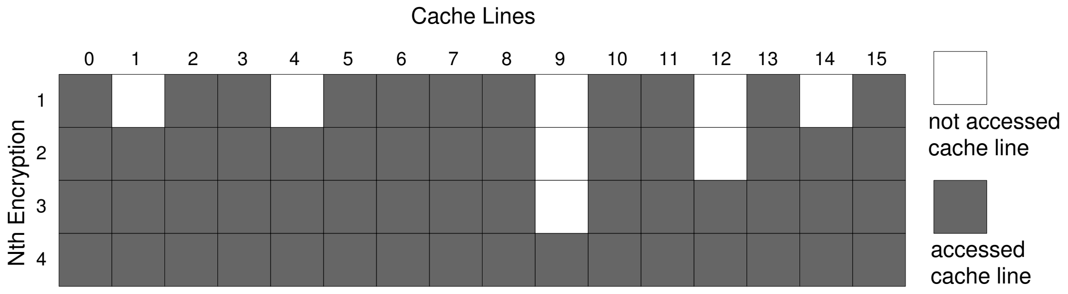


Figure 7: An example of using access-driven attacks to reveal cache line usage in the PLcache.

the relation between the key bytes. With 2^{14} samples, the derived relationship among different key bytes is enough for the tool to successfully recover the complete 128-bit key.

Besides the security vulnerabilities discussed above, the PLcache is also subject to denial-of-service attacks due to its inherent over-locking problem. Even though OS oversight was proposed to properly handle the locking requests or impose limit on the size of locked cache lines for each process, the PLcache design does not support the locked cache lines to be replaced when the owner process is not running, i.e. switched out because of context switches. As a result, processes may still lock too many cache lines, causing fairness problem or even severely degrading other processes' performance.

4.2 Analysis of RPcache

The main security issue with the RPcache is its vulnerability to cache-collision attacks. In cache-collision attacks, no interference is explicitly needed since their fundamental cause is that higher number of cache collisions lead to lower encryption time. As the RPcache does not guarantee that all the protected data (e.g., the AES lookup tables) are in the cache, this side channel still exists. We performed an experiment to test whether the RPcache is vulnerable to cache-collision attacks. In this experiment, we ran 16 million encryptions upon random 16-byte data with the same key and a clean cache state is setup before each encryption. Figure 8 shows the timing characteristics collected from our experiment. As shown in this figure, the relationship between the total number of cache collisions in the final round and the mean execution time still holds. The mean encryption time of the encryption runs with the correct key-byte difference is still significantly lower than the overall mean encryption time.

To get a solid proof, we implemented the RPcache in our timing processor simulator and conducted an actual cache-collision attack, the last round attack of Bonneau et. al. [11] on AES. Similar to PLcache results, 2^{14} samples were sufficient to completely recover 128-bit AES key on RPcache.

5. POTENTIAL SOLUTIONS

As analyzed in [25], the recently proposed secure cache architectures (i.e., the PLcache and the RPcache) have many desirable features including low performance overhead, generic countermeasure support, and transparency to the protected applications in the case of the RPcache. Unfortunately, as shown above, they are still vulnerable to software cache-based side channel attacks. Thus, in this section, we will briefly discuss potential solutions to their security problems.

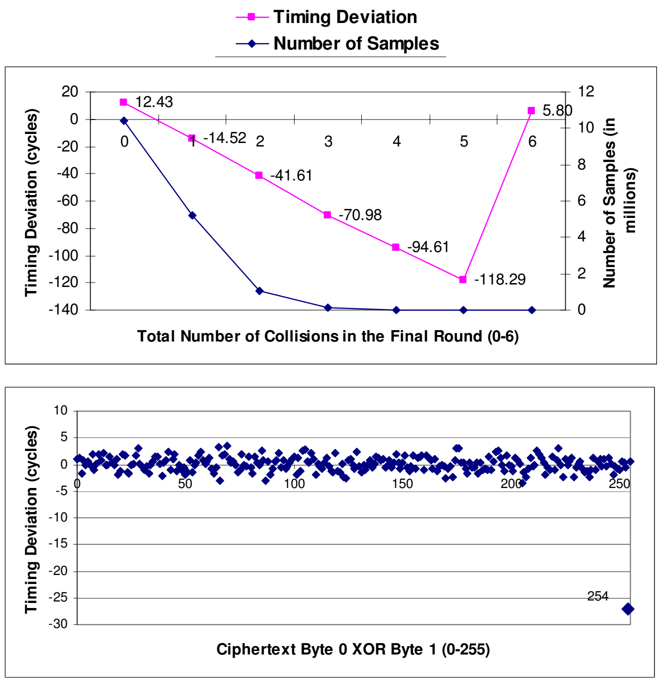


Figure 8: Experimental results of running last round cache-collision attack on RPcache.

5.1 Securing the PLcache

The main weakness of the PLcache design is that it does not protect the initial loading procedure. Therefore, to secure the PLcache, one possible solution would be a preloading and locking all critical data right before the crypto operations. This way, there is no initial loading process to be exploited for information leakage attacks. Note that this is different from pure software preloading. The reason is that without the PLcache support, pure software preloading can not guarantee that the critical data will not be replaced after the preloading process.

In terms of the denial-of-service vulnerability, the key is to make sure that when a process is not active, it will not lock its data in the data cache so that other processes will not suffer from the reduced cache capacity. Then, when an inactive process becomes active again, we need to reload and lock all its critical data since some of the data may be replaced by other processes' data. This way, OS oversight is not necessary for managing locking requests in uni-processors and each process can have locking request up to the whole cache without causing problems to other processes, which reduces the design complexity. However, in multi-threaded processors, there may be multiple active processes at the same time. OS oversight is still required to manage the processes that are competing for the locked cache lines. For example, OS may simply choose to run one single protected process at a time if an additional protected process requires overlapped locked cache lines.

There exist several possible ways to implement the mechanisms to secure the PLcache. One could be mainly software. This way, the protected process performs the preloading and locking before critical operations and performs unlocking once the critical operations are completed. The OS will perform the preloading and locking during context switches of the protected process. The preloading and locking can also be accelerated with hardware support. In this case, new instructions are required to specify the address and length of the critical data and to invoke the preloading and locking. Besides the security enhancement, these solutions need to be evaluated for performance overheads and complexity.

5.2 Securing the RPcache

The main weakness of the RPcache design is that it is vulnerable to collision-based time-driven attacks. The RPcache design defeats access-driven attacks by obfuscating cache interferences. However, it still leaks information due to the variations of the encryption/decryption time, which are dependent on the number of cache misses. Therefore, to defeat time-driven attacks, the ideal approach is to eliminate cache misses so as to remove the variations of the encryption/decryption time. However, since there is no locking mechanism in the RPcache design, there is no guarantee that the accesses to the critical data will be cache hits. As a result, both compulsory misses (i.e., initial access to the critical data) and conflict misses (re-access to the critical data after they are replaced) are possible. One potential mitigation technique would be to eliminate as many cache misses as possible so as to increase the number of required samples to an infeasible level.

To accomplish this idea, we need a way to reload the critical data after it is replaced. One symptom of replaced data is that the re-access will be a cache miss. To capture such events, we need to change the hardware architecture to expose the cache miss event to the software. Once the software is being notified by the critical event, certain countermeasures can be performed. A detailed study of how this approach will help to secure the RPcache design and how much performance overhead will be introduced is part of our future work.

6. CONCLUSIONS

In this paper, we analyzed the latest advances in the software cache-based side channel attacks. We practically demonstrated (by our MIPS simulations) that the previously proposed cache designs, the partition locked cache (PLcache) and random permutation cache (RPcache), although effective in defeating information leakage due to interferences, are vulnerable to the latest attacks built upon either cache sharing or cache collisions. Our results are particularly interesting as the authors of [25] proved a theorem which stated that the RPcache totally eliminates side-channel attacks due to cache line addresses Thus, as usual within the space of "provable security", our result shows again that any kind of "security proof" has to be carefully scrutinized for general validity or applicability. We also discussed the potential enhancement to overcome the vulnerabilities of these new cache designs.

7. ACKNOWLEDGMENTS

This research was initiated during Jingfei Kong's internship with Samsung Electronics. This work was supported in part by NSF CAREER award CCF-0747062.

8. REFERENCES

[1] O. Acıiçmez and Ç. K. Koç. Microarchitectural Attacks and Countermeasures. Chapter in *"Cryptographic Engineering"* by Ç. K. Koç, Springer, ISBN 0387718168, to be published in November 2008.

[2] O. Acıiçmez. Yet Another MicroArchitectural Attack: Exploiting I-Cache. *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, pages 11-18, ACM Press, 2007.

[3] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. On The Power of Simple Branch Prediction Analysis. *2007 ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS'07)*, R. Deng and P. Samarati, editors, pages 312-320, ACM Press, 2007.

[4] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. Predicting Secret Keys via Branch Prediction. *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, M. Abe, editor, pages 225-242, Springer-Verlag, Lecture Notes in Computer Science series 4377, 2007.

[5] O. Acıiçmez and J.-P. Seifert. Cheap Hardware Parallelism Implies Cheap Security. 4th *Workshop on Fault Diagnosis and Tolerance in Cryptography — FDTC 2007*, pages 80-91, IEEE Computer Society, 2007.

[6] O. Acıiçmez, W. Schindler, and Ç. K. Koç. Cache Based Remote Timing Attack on the AES. *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, M. Abe, editor, pages 271-286, Springer-Verlag, Lecture Notes in Computer Science series 4377, 2007.

[7] O. Acıiçmez, W. Schindler, Ç. K. Koç. Improving Brumley and Boneh Timing Attack on Unprotected SSL Implementations. *Proceedings of the 12th ACM Conference on Computer and Communications Security*, C. Meadows and P. Syverson, editors, pages 139-146, ACM Press, 2005.

[8] Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. Available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[9] AMD. Lightweight Profiling Proposal, AMD, July 2007. Available at: http://developer.amd.com/assets/HardwareExtensionsforLeightweightProfilingPublic20070720.pdf

[10] D. J. Bernstein. Cache-timing attacks on AES. Technical Report, 37 pages, April 2005.

[11] J. Bonneau and I. Mironov. Cache-Collision Timing Attacks against AES. *Cryptographic Hardware and Embedded Systems — CHES 2006*, L. Goubin and M. Matsui, editors, pages 201-215, Springer-Verlag, Lecture Notes in Computer Science series 4249, 2006.

[12] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. *Proceedings of the* 12th *Usenix Security Symposium*, pages 1-14, 2003.

[13] D. Burger and T.M. Austin. The Simplescalar Tool Set Version 2.0. Technical Report, Computer Science Department, University of Wisconsin-Madison, 1997.

[14] J. Daemen, V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.

[15] S. Gueron. Advanced Encryption Standard (AES) Instructions Set. Technical Report, 35 pages, Intel Corporation, April 2008. Available at: http://softwarecommunity.intel.com/isn/downloads/intelavx/AES-Instructions-Set_WP.pdf

[16] M. Neve and J.-P. Seifert. Advances on Access-driven Cache Attacks on AES. 13th *International Workshop on Selected Areas of Cryptography — SAC'06*, E. Biham and A. M. Youssef, editors, pages 147-162, Springer, Lecture Notes in Computer Science series 4356, 2007.

[17] OpenSSL Montgomery Exponentiation Side-Channel Local Information Disclosure Vulnerability. http://www.securityfocus.com/bid/25163/ 2007.

[18] OpenSSL Timing Attack RSA Private Key Information Disclosure Vulnerability. http://www.securityfocus.com/bid/7101/ 2003.

[19] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. *Topics in Cryptology — CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, D. Pointcheval, editor, pages 1-20, Springer-Verlag, Lecture Notes in Computer Science series 3860, 2006

[20] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.

[21] D. Page. Partitioned Cache Architecture as a Side Channel Defence Mechanism. Cryptography ePrint Archive, Report 2005/280, August 2005.

[22] C. Percival. Cache missing for fun and profit. *BSDCan 2005*, Ottawa, 2005.

[23] D. Song, D. Wagner, and X. Tian. Timing Analysis of Keystrokes and SSH Timing Attacks. *Proceedings of the* 10th *Usenix Security Symposium*, 2001.

[24] K. Tiri, O. Acıiçmez, M. Neve, and F. Andersen. An Analytical Model for Time-Driven Cache Attacks. 14th *International Workshop on Fast Software Encryption — FSE 2007*, A. Biryukov, editor, pages 399-413, Springer, Lecture Notes in Computer Science series 4593, 2007.

[25] Z. Wang and R. B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks, *the* 34th *International Symposium on Computer Architecture (ISCA 2007)*, June 9-13, 2007.