Hardware-Software Integrated Approaches to Defend Against Software Cache-based Side Channel Attacks

Jingfei Kong[1], Onur Acıiçmez[2], Jean-Pierre Seifert[3] and Huiyang Zhou[1]
University of Central Florida[1], Samsung Electronics[2], TU Berlin & Deutsche Telekom Laboratories[3]
{jfkong, zhou}@cs.ucf.edu[1], o.aciicmez@samsung.com[2], jpseifert@sec.t-labs.tu-berlin.de[3]

Abstract

Software cache-based side channel attacks present serious threats to modern computer systems. Using caches as a side channel, these attacks are able to derive secret keys used in cryptographic operations through legitimate activities. Among existing countermeasures, software solutions are typically application specific and incur substantial performance overhead. Recent hardware proposals including the Partition-Locked cache (PLcache) and Random-Permutation cache (RPcache) [23], although very effective in reducing performance overhead while enhancing the security level, may still be vulnerable to advanced cache attacks.

In this paper, we propose three hardware-software approaches to defend against software cache-based attacks - they present different tradeoffs between hardware complexity and performance overhead. First, we propose to use preloading to secure the PLcache. Second, we leverage informing loads, which is a lightweight architectural support originally proposed to improve memory performance, to protect the RPcache. Third, we propose novel software permutation to replace the random permutation hardware in the RPcache. This way, regular caches can be protected with hardware support for informing loads. In our experiments, we analyze various processor models for their vulnerability to cache attacks and demonstrate that even to the processor model that is most vulnerable to cache attacks, our proposed software-hardware integrated schemes provide strong security protection.

1. Introduction

Side channel attacks exploit "side channel" information such as power, heat, electromagnetic radiation, or time to derive confidential information, particularly secret keys used in cryptographic systems. Recently, there are newly developed software-based side channel attacks which exploit architectural features of modern commodity processors such as caches [1], [4], [5], [6], [15], [16], [19], [22] and branch predictors [2], [3]. These attacks do not require physical access to target computers or direct access to the memory space of victim processes and are conducted through legitimate software operations. As a result, they pose serious threats to modern computer systems [12, 23].

Current software cache-based side channel attacks include access-driven attacks [15], [16], [19] and time-driven attacks [4], [6]. Access-driven attacks exploit the correlation between the secret key and the cache usage of a crypto thread/process. Since the cache is shared among multiple processes/threads, an attacker may derive the cache usage of the victim process by controlling a carefully crafted process, which runs together with the victim process. Time-driven attacks measure the execution times of victim processes and exploit the correlation between the secret key and the number of cache misses (which in turn determines the execution time) to infer the key. To defend against software cache-based side channel attacks, various countermeasures have been proposed and many of them involve some modifications upon the software implementation of crypto algorithms [7], [17], [26]. However, these proposals are often application and attack specific. In order to achieve a high level of security protection, several defense techniques need to be combined, resulting in substantial performance overhead. In a recent work [23], Wang and Lee identify certain features in data caches as the root cause for software cache-based side channel attacks, and propose new cache designs (PLcache and RPcache) to prevent information leakage. Such hardware-based defenses, although effective for their targeted attacks, lack the flexibility to adapt to newly developed attacks [14]. Another approach to defeat cache-based attacks is to dedicate special hardware function units and instructions to a particular crypto algorithm, such as Intel's AES (Advanced Encryption Standard [10]) instructions [11], so that cache accesses can be completely eliminated during crypto operations. This approach, however, requires non-trivial hardware and software changes since existing crypto software has to be re-written/recompiled to leverage the new AES instructions. Furthermore, it does not protect crypto algorithms other than AES.

In this paper, we review the state-of-art cache attacks and identify that in both access-driven and timing-driven attacks, cache misses of critical data, whose addresses are dependent on secret keys, are the source of information leakage. We then propose three integrated hardware-software mitigation approaches. First, we propose to use preloading to secure the previously proposed PLcache [23] so as to ensure that all accesses to critical data will be cache hits. Second, we propose to use informing loads to protect the RPcache [23]. Informing loads [13] are lightweight architectural support originally proposed for

978-1-4244-2932-5/08/$25.00 ©2008 IEEE

optimizing memory system performance. When an informing load (a special load instruction) misses in the cache, a user-level exception is raised. With the support for informing loads, we can easily integrate flexible software-based mitigation schemes into exception handlers. Although the RPcache randomizes cache miss addresses through random permutation, it is vulnerable to time-driven attacks (see Section 3.2) and our software scheme fixes the vulnerability by re-loading all critical data upon cache miss detection. It ensures that all subsequent cache accesses to those data are cache hits as long as the critical data fits in the cache and thus removes the correlation between the secret key and the number of cache misses. Third, we propose a software permutation scheme assisted by informing loads to replace the random permutation logic in the RPcache. This way, the protection can be extended to regular caches with the relatively minor hardware support for informing loads. Our experiments show that the proposed approaches based on the PLcache and RPcache provide strong protection with low performance overhead. For regular caches, our lightweight informing loads approach not only provides strong security protection without the high hardware cost of the PLcache or RPcache, but also has significantly lower performance overhead compared to existing software-only solutions.

The remainder of the paper is organized as follows. In Section 2, we define the threat model and analyze the software cache-based side channel attacks. Section 3 discusses existing software and hardware countermeasures. In Section 4, we propose integrated hardware-software approaches to defend against cache attacks. Section 5 demonstrates that our solutions are effective against cache attacks and examines their performance overheads. We summarize the paper in Section 6.

2. Threat Model and Attacks

There exist mainly two types of software cache-based side channel attacks: access-driven and time-driven attacks. In access-driven attacks, the adversary has control over one or multiple spy processes, which share the cache with the victim process. Due to cache sharing, the victim process may evict the spy process' cache lines when it accesses key-dependent (i.e. critical) cache lines. By measuring the access times of its own cache lines, the spy process can figure out which cache lines are evicted by the victim process. Such cache access behavior of the victim process may leak enough information for the adversary to infer the key. In time-driven attacks, the adversary sends various encryption/decryption requests to the target crypto process. Upon receiving responses the adversary records the encryption times. Since the secret key may correlate to different number of cache misses upon different inputs/outputs, the variations among encryption times may provide sufficient information for the adversary to derive the key. Although time-driven attacks may be much slower than access-driven attacks, we consider both in this paper.

In this paper, we use one widely used cryptographic algorithm – the Advanced Encryption Standard (AES) [10] to illustrate current cache attacks as well as the existing countermeasures and demonstrate the advantages of our proposed schemes. However it should be noted that our proposed schemes may also be applied to cache attacks on other applications.

2.1 The Advanced Encryption Standard (AES)

AES processes a 16-byte input with a secret key of 16, 24 or 32 bytes to produce a 16-byte output. There are multiple identical rounds involved in encryption/decryption and each round performs four types of operations (substitute bytes, shift rows, mix columns and add round key). Among them, the "substitute bytes" operation requires table lookups, in which a 1-byte input is used as an index to a compact S-box table (an 8-bit substitution box) to generate a 1-byte output. For fast software AES implementations, the four operations are combined into 16 XOR operations and 16 table lookups. The tradeoff is that five new lookup tables (T_0, T_1, T_2, T_3 and T_4) are used with each having 256 4-byte elements, larger than the original S-box table with 256 1-byte elements [10].

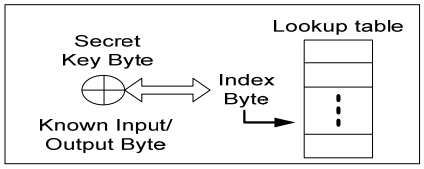

Fig. 1. Vulnerable table lookup operations in AES

2.2 Access-driven Attacks against AES

In AES, two components decide the indices of table lookups. One is the 16-byte input and/or output and the other is the secret key (as shown in Figure 1). As a result, the key can be computed if an adversary obtains both the input/output and the indices of table lookups. Since the output of AES (i.e., the encrypted ciphertext) is not kept private and/or sometimes the adversary may even know the plaintext (i.e., the input to AES), it is reasonable to assume the availability of the input/output. So, the critical step for the key recovery is to obtain the indices of table lookups. Since the indices determine which cache lines are accessed in the shared cache, the adversary is able to recover the key once the accessed cache lines can be identified. To identify the cache lines accessed by AES table lookups, access-driven attacks require that those cache lines *must not* reside in the shared cache beforehand so that some of the spy process' data can be replaced later. In other words, the table lookups shall experience cache misses. This condition is essential for the access-driven attacks otherwise the spy process cannot know the cache usage. Such condition, however, can be satisfied by

the spy process, which loads a large data set to effectively flush the cached data of the victim crypto process.

There are some complications regarding the realization of access-driven attacks. For example, one cache line may contain several lookup table elements and thus knowing one accessed cache line does not exactly lead to the corresponding index value. Nevertheless, from multiple samples (as few as 15 [15]) the correct key values can be filtered out statistically. Detailed description of access-driven attacks against AES is reported by Neve [15] and Osvik [16].

2.3 Time-driven Attacks against AES

As discussed in Section 2.1, AES relies heavily on table lookup operations and the indices to lookup tables depend on the key and inputs/outputs. If prior to AES execution the lookup tables are not in the cache, different data inputs may cause different sequences of table lookups, which in turn result in different numbers of cache misses and thus different execution times. Time-driven attacks exploit the relationship between inputs/outputs and execution times to infer the key. Bernstein [4] demonstrated that different inputs can cause various execution times and thus the key can be inferred. Bonneau [6] presented a cache-collision attack that finds the key using a much smaller number of timing samples.

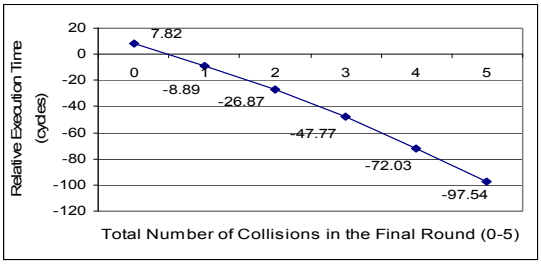

Fig. 2. The relationship between the number of collisions in the last round of AES and the encryption time on one Pentium 4 machine.

A cache collision happens when two table lookups refer to the same element, in which case, the second lookup will be a cache hit assuming no conflict misses occurred in between. For a non-cache-collision case, the second lookup may experience a cache miss. For successful cache-collision timing attacks on AES, the basic observation is that a higher number of cache collisions results in a smaller number of cache misses, thus a shorter encryption time [6],[22]. For example, Figure 2 shows the relationship between the mean execution time of one AES encryption and the number of cache collisions in the last round of AES. The results are collected from 16 millions timing samples using one Pentium 4 machine running AES with the same random key. As shown in the figure, the execution time (relative to the average of overall timing samples) declines as the number of collisions increases (e.g., +7.82 cycles for 0 collision and -8.89 cycles for 1 collision). Based on this observation, attackers can pick two table lookups and guess cache-collision cases to infer the XORed values between key bytes, which lead to the complete key recovery. In this particular example, 2^{15} samples were good enough for revealing the key. Detailed information on cache-collision attacks can be found in [6], [22].

2.4 Source of Information Leakage in Software Cache-based Side Channel Attacks

Although access-driven attacks and time-driven attacks are different in the way the attacks are performed, the source of information leakage is the same: cache misses of lookup table data (whose indices are key dependent) are exploited to infer the key. Section 4 shows how we utilize this observation to defend against both types of cache attacks.

3. Current Countermeasures

Current countermeasure proposals are either application-level software solutions or hardware solutions. Among software approaches, each vulnerable application is analyzed and changed against specific attacks. In order to achieve reasonable security protection, software approaches are often combined, which may incur substantial performance overhead. Hardware proposals revise cache architecture to eliminate information leakage exploited in cache attacks. Although hardware schemes are able to provide general protection at small performance cost, they often incur non-trivial hardware changes and suffer from inflexibility to evolve against newly developed attacks.

3.1 Software Countermeasures

(1) Access-all against access-driven attacks

Since cache attacks use cache lines to infer the indices to the lookup tables, one way to defeat them is to eliminate the correspondence between cache lines and table indices. In a released patch [26] of RSA against access-driven attacks [19], each table element is distributed so that accessing one element ends up accessing all the cache lines of the whole table. This approach is effective for RSA (only less than 10% performance overhead) given the heavy computations involved in RSA. However, it is not applicable to AES since touching the entire table for each table lookup operation incurs too much performance overhead given the high number of table lookups in AES.

(2) Random permutation against access-driven attacks

Random permutation of lookup tables changes the mapping between table indices and cache lines. It obfuscates attackers' observation on cache access activities. However fixed permutation can still leak information, as demonstrated in [5] for AES. Although the security offered by random permutation can be increased by frequently updating the permutation, the updating frequency remains an open question for pure

software approaches due to the tradeoff between performance and security.

(3) Small tables against access/time-driven attacks

Efficient AES implementations typically utilize large lookup tables, which are pre-computed from the original S-box table. One protection scheme is to use the small S-box table instead to trade performance for security [16]. This way, one single cache line contains more elements, complicating the attacks. There are also other similar approaches, which use smaller numbers of lookup tables [16]. However, at the cost of substantial performance overhead, these approaches only increase the number of samples required for access-driven and time-driven attacks [5],[22].

(4) Preloading against access/time-driven attacks

Preloading of all lookup tables before cryptographic operations aims to mask cache access activities to prevent cache attacks [17]. However, preloading still provides no security guarantee against access-driven attacks since the adversary may still use spy processes to evict the lookup tables after the preloading process.

(5) Hybrid approaches

In [7], several defense techniques are combined to provide secure and efficient protection for AES.

***** *Software version 1* (v1): All rounds use the compact S-box table without permutation. Preloading is performed before each round. This is a combination of (3) and (4).

***** *Software version 2* (v2): The most vulnerable rounds (first round and last round) use the compact S-box table. The other rounds use large pre-computed lookup tables. All tables are permuted. Preloading is performed before the first round and the last round. It is a combination of (2), (3) and (4) and it trades security for improved performance.

***** *Software version 3* (v3): All rounds use the compact S-box table with permutation. Preloading is performed before each round. This is a combination of (2), (3) and (4).

Among the three, the v3 is most secure as it combines all three mitigation techniques. The v2 is most performance efficient as it uses large pre-computed lookup tables in inner rounds and does not preload them.

3.2 Hardware Countermeasures

Realizing the limitations of software countermeasures, several hardware schemes are proposed to provide comprehensive, efficient, and generic solutions (i.e., not application/attack specific) to defend against cache attacks.

3.2.1 Partitioned cache and Partition-Locked cache (PLcache)

In partitioned caches [18], a part of the cache is allocated exclusively to the protected process in order to prevent information leakage. This may cause inefficient cache sharing since the cache partition is fixed statically. In a recent work, the partition-locked cache (PLcache) [23] is proposed to address this problem with a fine-grained locking control so that only the cache lines, which contain the critical data, are isolated. The hardware support for the PLcache includes two additional fields in each cache line: an ID field and a lock bit. The ID indicates the owner of the cache line, normally a process and the lock bit indicates the locking status of the cache line. As for control interface, two mechanisms are proposed: ISA extension and segment/page-based protection. The first introduces several new instructions to provide fine-grain locking control of the cache lines. The second involves new OS-level API calls for coarse-grain control of memory regions. Also the cache line replacement policy is changed to support the locking mechanism.

3.2.2 Random-permutation cache (RPcache)

In contrast to partitioned caches, the random-permutation cache (RPcache) [23] allows flexible cache sharing but randomizes the mapping between memory addresses (i.e., table indices) and cache lines to prevent information leakage. In the RPcache, in case of cache interference, i.e. when the fetched cache line and the chosen replacement cache line belong to two different processes, the original cache set will not be used for replacement. Instead, another cache set is chosen randomly and replacement happens in that set. This changes the mapping between addresses and cache sets. Because of the swapping of cache sets, the cache lines in the original sets are invalidated. The hardware support includes a permutation table and a revised replacement policy.

3.2.3 Security issues with the PLcache and RPcache

The PLcache can still be vulnerable to both types of cache attacks since AES may still experience cache misses over the critical data before all of them are fetched and locked in the cache. While software can be used to pre-load the AES tables, such initial loading of the critical data may still provide enough information leakage for key recovery. Besides, the PLcache does not support locked cache lines to be replaced even when they are not needed (i.e. the owner process is switched out and not active). This may cause excessive locking (unless properly controlled by the OS [23]), and in any case reduces the size of the cache available to other processes. The RPcache defeats access-driven attacks because even if an adversary knows which cache lines are accessed by a crypto operation, the corresponding index can not be derived due to random permutation. The vulnerability of the RPcache, however, lies in its inability to defend against cache-collision time-driven attacks since random permutation does not eliminate the execution time variances: a high number of collisions still results in lower execution times. More detailed security analysis and examples of successful attacks to the PLcache and RPcache are presented in [14].

4. Integrated Hardware-Software Protection Schemes

In this section, we propose three hardware-software approaches to eliminate the source of information leakage exploited in software cache-based attacks. First, we propose to secure the PLcache by pre-loading the critical data (e.g. the AES lookup tables). Second, we advocate using informing loads to protect the RPcache from time-driven attacks. With the support for informing loads, we are able to respond to the source of information leakage – caches misses over the critical data and integrate flexible software-based defenses. Third, we propose informing loads assisted software-based random permutation to replace the permutation logic in the RPcache so as to provide the security protection to regular caches. These three approaches present different tradeoffs between hardware complexity and performance overhead.

4.1 Preloading to Protect Partition-Locked cache

4.1.1 The idea

Previous works [5], [7], [17] discussed the concept of preloading or cache warming as a possible countermeasure for cache attacks. The basic procedure is to load all security critical data, e.g., the AES lookup tables, into the cache right before the crypto operations. Preloading itself, however, cannot provide sufficient protection against cache attacks simply because an adversary can still manipulate the cache state after the preloading process. As discussed in Section 3.2, the PLcache does not provide high security either due to the initial loading process of the critical data. However, combining preloading and PLcache provides a solid protection mechanism against access/time-driven attacks.

The key here is to make sure that before cryptographic operations all the critical data are preloaded and locked in the PLcache. After that, any access to those critical data will result in a cache hit. This way, time-driven attacks are effectively defeated since there is no correlation between the secret key and cache hit/miss patterns. This scheme also defeats access-driven attacks since all a spy process can observe is that the whole set of the critical data are in the cache, thereby leaking no information of which parts of the critical table (or the indices to the tables in AES) are used in a crypto operation. Here, note that the PLcache with preloading is secure only if all the critical data can fit in the cache. This issue is generally not a problem for cryptographic algorithms, of which a key design objective is to keep critical data small [10]. The five critical lookup tables in AES, for example, take 5KB (1KB for each table).

To address the issue of excessive locking associated with the PLcache, we propose to allow the locked cache lines to be replaced if the cryptographic process is switched out (i.e., not active). When the cryptographic process is switched back, those protected cache lines will be reloaded and locked again.

4.1.2 Proposed implementation

Changes to the PLcache logic The change to the PLcache logic is that when a new non-protected cache line is about to replace a locked (protected) cache line, which is always prohibited in the PLcache, the replacement is now allowed if the owner process of the locked cache line is not active. This is done by comparing the ID field of a locked cache line with the active processes' IDs.

Architectural support for preloading One implementation of preloading and reloading is through hardware logic. In this implementation, a new preloading instruction will be used to specify the beginning address and length of the protected data. The preloading state (i.e., whether the preload has been performed or not and the address range of preloading) becomes part of the process context. Then, if the preloading state is set after a context switch, the hardware will re-load all critical data. This pure hardware approach may be too costly and introduce extra hardware complexity.

The preloading and reloading can also be implemented through software in an un-modified PLcache system. The protected process performs the preloading and locking before critical operations and performs unlocking once the critical operations are completed. The operating system (OS), however, needs to be changed to perform the preloading and locking during context switches of the protected process. Instead, we propose to use user-level exception handling to provide efficient preloading and offer flexibility to deploy newly developed software defense mechanisms.

User-level exception handling is introduced by Thekkath et al. [21]. It provides efficient handling of synchronous exceptions by user-level code. In our design, we implement it in the way described in [13]. When some instruction triggers a user-level exception, it works as a conditional branch. The exception only changes the program counter (PC) of the running process. It does not invoke any OS code. Necessary additions include two new instructions (EH-register and EH-jr) and two new registers (an exception handler address register-EHAR and an exception handler return register-EHRR). EH-register will load the entry address of the user-level exception handler to the EHAR. EH-jr is used at the end of the exception handler to jump to the address stored in EHRR to resume program execution. The procedure works as follows. The user-level exception event will trigger a pipeline squash when the offending instruction reaches the head of the Reorder Buffer (ROB). The next PC is saved to EHRR. Then the exception handler whose address is stored in EHAR will take over the execution and run as a regular function. It saves the registers that it will use to the stack at the beginning and restores those registers at the end. When the handling finishes, EH-jr will use EHRR to return to the interrupted process.

Besides the support for the user-level exception handling mechanism, we propose two new instructions (PL-begin and PL-end) and one new control status

register (PL-S). PL-begin and PL-end are inserted to enclose the cryptographic operations that need to be protected. The status register PL-S indicates the state of locking and preloading.

The whole procedure works as follows:

1) The user-level exception handling mechanism registers the entry address of the user-level exception handler during program initialization.

2) When the crypto process comes to the cryptographic operation, PL-begin sets the status register (PL-S) to indicate that the PLcache locking mechanism and preloading become effective for this crypto process. In the meanwhile, it triggers the exception handler, which pre-loads and locks all the critical data.

3) During program execution, if a context switch happens, the status register PL-S will be saved along with other states of the process. The PLcache locking mechanism becomes ineffective for the switched-out process, by comparing every cache access with an Active Process register. When the process is switched back, the status register PL-S is examined. If PL-S indicates that the critical data needs to be loaded and locked, a user-level exception is raised and the same exception handler will reload and lock the critical data to the cache.

4) When the protected cryptographic operations are completed, the PL-end instruction resets the status register, indicating that the critical data are no longer needed. The PLcache locking mechanism becomes ineffective for the process.

4.2 Securing the RPcache with Informing Loads

4.2.1 The idea

With cache-based attacks taking advantage of cache hit/miss behavior of crypto processes, we argue that the crypto process itself can leverage the same information to defend against the attacks. Informing loads, originally proposed as a lightweight architectural support for memory optimization [13], enable the crypto process to gain the control once an access to critical data misses in the cache. This way, flexible software defense mechanisms can be deployed. In this section, we show that informing loads can be used to effectively address the security vulnerability of the RPcache. Because of its flexibility, our approach can be further extended to protect regular caches (Section 4.3).

As explained in Section 3.2, the RPcache defeats access-driven attacks by randomizing cache line mapping. However, it is vulnerable to cache collision attacks since not all critical data are guaranteed to reside in the cache. Therefore, the fundamental assumption behind collision attacks still holds. In other words, a higher number of collisions still results in lower encryption time. To protect the RPcache, we propose to use informing loads to access the critical data. In the case of AES, it means that all the table lookups are implemented using informing loads while other data accesses use regular load instructions. The informing loads detect whether the critical data is in the cache. If not, they will redirect the PC to a user-level exception handler. Note that preloading alone can't provide sufficient security support against cache collision attacks as preloaded data can still be replaced. Such a case could be that during the execution, internal data of the victim process may replace the preloaded data, which is identified as internal interference in [23].

In this paper, we devise an exception handler to load all the critical data (or the tables T_0-T_4, in AES) into the cache. The objective is that after the exception handling, subsequent accesses to the critical data will hit in the cache, thereby eliminating time variations. Note that, such data loading is just one possible solution. The key advantage of using informing loads over pure hardware-based defense mechanisms such as the RPcache is that the exception handler can be easily updated to defend/detect future cache-based attacks or to incorporate a better crafted defense algorithm.

4.2.2 Proposed implementation

Informing loads are special load instructions that "inform" the software when the load misses in the cache. There are three ways of implementing informing loads [13]. The first is to use a cache outcome condition code and branch-and-link instructions. The second is a branch operation with a slot that is squashed if there is a hit. The third one is a low-overhead user-level cache miss trap. We choose to use the low-overhead cache-miss trap for its low hardware complexity.

```
r = random_number;
// from hardware random number generator (i.e. the one
// used by the RPcache)
 i_max = number_of_tables;
 j_max = table_size / table_element_size;
 for (i=0; i < i_max; i++)  // Fetch each protected table
   for (j=0; j < j_max; j +=
                        cache_line_size/table_element_size)
       Prefetch( T[(i XOR r) % i_max][(j XOR r) % j_max] );
       // Fetch each protected cache line in a random order
```

Fig 3. Code of the informing load exception handler

Using AES as an example, the proposed procedure works as following. In the cryptographic operations, those protected tables are loaded with informing load instructions. These informing load instructions work as normal loads with no extra overhead when they hit in the cache. Whenever an informing load misses in the cache, a user-level exception will be generated and the exception handler, shown in Figure 3, will be executed. As shown in Figure 3, the exception handler loads the critical tables in a random order. The reason is due to an artifact of the RPcache, in which cache lines may be invalidated when a cache line index is randomized. As a result, if the exception handler loads the tables in a determined order, the elements that are loaded earlier have higher chances to be invalidated than those loaded later. The random order (achieved by *XOR*ing a random number *r* in Figure 3) in the exception handler eliminates the determinism.

Combining informing loads with the RPcache defeats software cache-based attacks. The RPcache itself is

effective against access-driven attacks and the informing load exception handling provides a defense mechanism against time-driven attacks. Due to the invalidation effect of the RPcache, in theory we still can not guarantee that all the table access hit in the RPcache since some critical data may be invalidated during the permutation process. As a result, there may still be access latency variations among different table elements, although the randomized loading order in our exception handler already makes such variations non-deterministic. To completely overcome the problem, we can change the RPcache to let it selectively swap the cache lines instead of invalidating them during the permutation process. In other words, if a cache line to be invalidated contains some critical data, the content will be copied to the new location instead of being invalidated. Such a change increases the complexity of the RPcache but may further improve the security. With such implementation, the randomized loading order in the informing load exception handler can be removed.

4.3 Securing Regular Caches with Informing Loads

4.3.1 The idea

To protect regular caches, we first use software random permutation to randomize the crypto process' cache footprint against access-driven attacks. However, as discussed in Section 3.1, fixed permutation leaks information and frequent updates of permutation may incur unnecessary performance overheads. To overcome these problems, we propose to change the permutation only when it is necessary. As discussed in Section 2, cache attacks rely on cache misses to identify whether a cache line is used by the victim process. Therefore, we choose to change the permutation whenever there is a cache miss of the critical data, which is supported by informing loads. In other words, we integrate permutation update in the exception handler of informing loads. Such informing loads assisted software permutation can also be viewed as a replacement of the permutation logic in the RPcache so that the security protection can be provided with no need for hardware changes for the RPcache.

To defeat time-driven attacks, loading of all critical data is also performed in the exception handler upon cache misses detected by informing loads, similar to the way to further secure the RPcache.

4.3.2 Proposed implementation

The key of software permutation is to use one level of indirection to randomize the address mapping between the table lookup values and their memory addresses. Using the critical tables of AES as an example, we use an indirection table to produce the actual address of one protected unit, i.e. a cache line in our implementation, as shown in Figure. 4.

In Figure 4, one dimension table T with N elements is converted into a two-dimension K x L array T', where K x $L = N$ and L is the number of elements in one cache line, i.e. L = *Cache line size / size of each table element*. Compared to T, which occupies a continuous region of memory, the data in T' are distributed in memory and are accessed through pointer indirection.

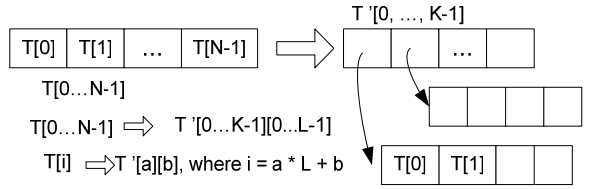

Fig. 4. Converting a one-dimension table into a two-dimension array.

With the critical data organized in a two-dimension array, we can easily perform address permutation. As illustrated in Figure 5, assuming that the address of table element T[0] (i.e., &T[0] or T'[0]) is 0x40, address permutation upon T'[0] proceeds as follows. First, one entry among T'[1, … K-1] is randomly selected (assuming T'[1] is selected and T'[1] = 0x80). Second, both the indirection pointers (T'[0] and T'[1]) and the data pointed to (*T'[0] and *T'[1]) are swapped. After permutation, the new address of T[0] becomes 0x80. The data value of T[0] (or T'[0][0]) is unchanged due to the data swapping between *T'[0] and *T'[1].

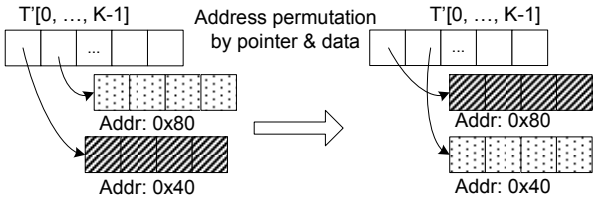

Fig. 5. Address permutation by swapping both the pointers and the data.

With the proposed way to perform permutation, we devise the exception handler for informing loads to defeat the cache attacks. The scheme works as follows. In the crypto algorithm (e.g., AES) implementation, two-dimension arrays are used to store the critical data (each of T_0-T_4 in AES). The table lookups use informing loads while other data accesses use regular load instructions. Once a critical data access misses in the cache, the exception handler for informing loads will be triggered. The exception handler prefetches all critical data to the cache and meanwhile performs the permutation change upon the missing table entry, as illustrated with the pseudo code in Figure. 6.

```
0. The cache line of T[i] is missing in the cache
1. Prefetch from addresses T'[0],T'[1],…,T'[K-1]
2. Find the corresponding T'[p] that points to T[i]
3. Randomly select a target entry T'[q] for permutation
4. Swap *T'[p] and *T'[q]; swap T'[p] and T'[q]
```

Fig. 6. Pseudo code of the exception handler for informing loads, which prefetches all critical data to the cache and randomly permutes the cache lines.

5. Experiments

5.1 Methodology

Our experiments are conducted using a detailed timing simulator developed from the SimpleScalar toolset [8]. The underlying processor model is MIPS R10000 and the default configuration is listed in Table 1. We implemented both the PLcache and RPcache in the simulator. Proper architectural support for informing loads is included in the simulator. The detailed user-level exception handling including pipeline squashing, control flow transfer to and from the user-level handler is implemented to faithfully measure the performance of our proposed approaches.

Table 1. Default processor configuration

Branch Predictor	64K-entry g-share, 4K-entry direct mapped Branch Target Buffer (BTB)
Superscalar Core	7-stage pipeline: Fetch/Dispatch/Issue/RegisterRead/EXE /WriteBack/Retire, Pipeline bandwidth:4
	Fully-symmetric Function Units: 4
	Reorder Buffer (ROB) size: 128 Issue Queue (IQ) size: 64 Load Store Queue (LSQ) size: 64
	Fetch Policy for SMT: round-robin
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 8 Miss Status Handling Registers (MSHRs)
L2 Unified Cache	2MB 16-way Block size: 64B 300-cycle miss penalty

Our AES code is extracted from the OpenSSL 0.9.7c implementation. The key size is 16 bytes. For performance evaluation, the OpenSSL speed test program, a standard microbenchmark program included in OpenSSL [24], is used as our benchmark program. In the program, AES runs in the cipher-block chaining (CBC) mode. We use the message size of 8KB in our experiments. In terms of the performance metric for AES, we use cycles per byte instead of instructions per cycle. The reason is that different hardware/software approaches have different implementations, different lookup tables or different number of instructions. Cycles per byte, in contrast, directly reflects the throughput of AES: the number of bytes encrypted per time unit.

5.2 Security Analysis

5.2.1 Microarchitectural effects on cache collision time-driven attacks

We first investigate the microarchitectural effects on cache-collision time-driven attacks against AES. The reason is that both instruction-level parallelism (ILP) and memory-level parallelism (MLP) affect the encryption time. With a high degree of parallelism, many cache misses can be overlapped, thereby reducing the impact from cache collisions. In this experiment, we use five processor configurations as shown in Table 2 and examine the relationship between the encryption time and the number of cache collisions. In the experiment, a clean cache state is established before the 128-bit-key standard OpenSSL AES encryption and 16-million samples are collected for each processor configuration. Since different processor configurations lead to different encryption times, we report the relative encryption times in Figure 7, in which the encryption times are normalized to the mean encryption time with the same configuration. For reference, we also include the results from a real Pentium 4 machine in Figure 7.

Table 2. Different processor configurations to evaluate cache collision effects

Configuration 1	In order issue, 1way-issue, 1MSHR
Configuration 2	In order issue, 4way-issue, 4MSHRs
Configuration 3	Out-of-order execution, 4way-issue 4MSHRs, 64ROB/32IQ/32/LSQ
Default Configuration	Out-of-order execution, 4way-issue 8MSHRs, 128ROB/64IQ/64/LSQ
Configuration 4	Out-of-order execution, 8way-issue 32MSHRs, 256ROB/128IQ/128/LSQ

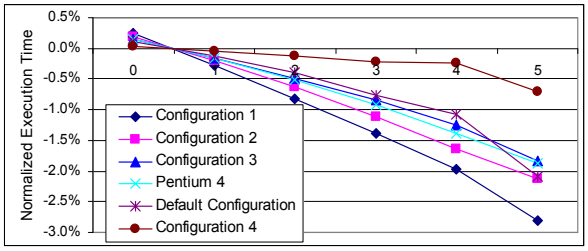

Fig. 7. The relationship between the number of collisions in the final round and the normalized encryption time on various processor configurations.

From Figure 7, it can be seen that the collision attacks are most effective against single issue, in-order processors with limited MLP. Both out-of-order (OOO) execution and high degrees of MLP reduce the effect of cache collisions. Furthermore, we perform AES final-round collision attacks on those samples. The analysis tool that we used is from [6], which performs collision attack analysis (i.e., key search) on AES encryption samples using a number of artificial intelligence techniques and reports the number of samples required to recover the complete key. The results, which are shown in Table 3, validate our observations. For processor configuration 1, around 4k samples are enough to break the key. In comparison, it takes around 400k samples to break the key for processor configuration 4. Our experiments on these processor configurations with the RPcache also show similar results for the different processor configurations, confirming that the RPcache is still vulnerable to collision attacks. From this experiment, it can also be seen that encryption times of modern high performance processors is less correlated to the number of cache collisions (or the number of cache misses) than in-order processors. This is due to the effect of OOO

execution and MLP. However the vulnerability to collision attacks remains since the number of samples required to reveal the correlation is still in feasible ranges.

Table 3. Required numbers of samples for key recovery on various processor configurations. (based on 25 random keys with random input plaintext)

	min	median	max
Configuration 1	3k	4k	5k
Configuration 2	6k	10k	13k
Configuration 3	7k	11k	13k
A real Pentium 4	12k	20k	27k
Default Configuration	17k	24k	29k
Configuration 4	328k	393k	459k

5.2.2 Security evaluation of the proposed schemes

The PLcache with preloading (PLcache+PL) As discussed in Section 4.1, the PLcache with preloading ensures that all critical data reside in the cache throughout the crypto process' lifetime. It defeats access-driven attacks since the only information that a spy process/thread can obtain is that all the critical data are used. It defeats time-driven attacks as any access to the critical data will hit in cache, thereby no timing variations.

The RPcache with informing loads (RPcache+IL) Through random permutation, the RPcache is effective against access-driven attacks (see [23] for the theoretical proof). With informing loads, any access to the critical data, if it misses in the cache, will invoke the exception handler to load all the critical data. Next, we examine how well this approach mitigates cache-collision attacks.

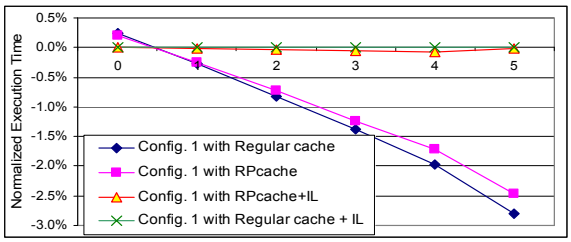

Fig. 8. The effect of our informing loads approaches on the relationship between the number of collisions in the final round and the normalized encryption time.

We first examine the relationship between the encryption time and the number of cache collisions as shown in the Figure 8. As analyzed in Section 5.2.1, processor configuration 1 is most vulnerable to collision attacks. Therefore, we use this processor configuration to evaluate various cache designs. For the RPcache enhanced by informing loads (Config. 1 with RPcache+IL), we observed no evident correlation between the encryption time and the number of cache collisions compared to the RPcache case (Config. 1 with RPcache). Then, we repeated the key recovery experiments. Compared to the RPcache, upon which 8K samples are enough for a complete key recovery, the attack fails on the RPcache+IL when presented with 16-million samples. More samples have not been tried due to the required simulation time as it takes 16 days to simulate one 16-million encryption run. Attacks on other processor configurations equipped with the RPcache+IL also failed when presented with 16-million samples.

From the experiments, we can conclude that using informing loads with the proposed exception handler can greatly enhance the security of the RPcache. However, as discussed in Section 4.2.2, if further cache security is desired, one can choose to selectively swap cache lines instead of invalidating them. This way, the cache collision attacks upon uniprocessors can be completely defeated since all the tables reside in the cache and all the access to those tables are cache hits. For more complex cache-collision attacks upon multi-threaded processors, further exploration is necessary and left as our future work.

Regular caches with informing loads (Regular cache+IL) For access-driven attacks, our solution is to use software permutation to randomize the mapping of the cache lines that contain critical data. Every time if the adversary tries to exploit a cache miss to observe the cache usage, the permutation varies. In fact, the permutation with updates on cache misses can be viewed as a software implementation of the RPcache design, which is already proven to be secure from access-driven attacks from the information theory perspective [23].

For time-driven attacks against AES, again we examine the relationship between the encryption time and the number of cache collisions as in the previous sections and the results are also in Figure 8. From Figure 8, it can be seen that for processor configuration 1, there exists no evident correlation between the encryption time and the number of cache collisions in our informing loads approach (Config. 1 with Regular cache+IL) compared to the regular cache case (Config. 1 with Regular cache). Next, we repeat the cache-collision attacks to evaluate the effectiveness of our approach against time-driven attacks on the processor configuration 1. For the regular cache the tool successfully retrieves the key with less than 8K samples. For the cache protected by our approach, the tool fails on 16-million samples, demonstrating our approach's effectiveness. Attacks on other configurations equipped our regular cache+IL also failed on 16-million samples. This is similar to what we observed from the RPcache+IL, as both use reloading of all the critical data as the countermeasure against collision attacks.

5.3 Performance Evaluation

In this section, we study the performance impact of our proposed approaches, i.e., preloading on top of the PLcache (PLcache+PL), informing loads combined with the RPcache (RPcache+IL) and informing loads combined with the regular cache (Regular cache+IL), upon AES, which represents the code to be protected. The baseline is the standard OpenSSL AES implementation which uses 5 precomputed lookup tables with the total size of 5KB.

5.3.1 Performance impact on AES

In this experiment, we analyze the performance impact of different protection schemes on AES using various L1 data cache configurations. We vary the L1 data cache

sizes from 8KB to 32KB and the set associativity from 1-way to 4-way. The throughputs normalized to the baseline results (i.e., regular cache) are shown in Figure. 9. Among different cache designs, the PLcache and PLcache+PL have almost the same performance since all critical data are locked in both designs after the short warm-up phase.

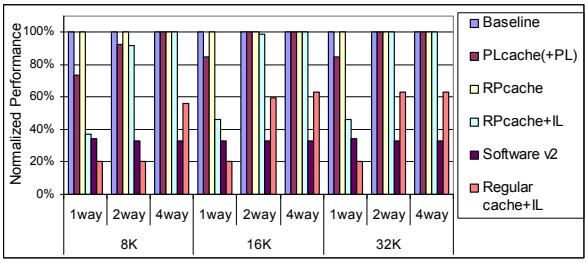

Fig. 9. Performance impacts of different protection schemes over various cache configurations.

From Figure 9, we make the following observations. First, for the 8KB direct-mapped cache, the PLcache (and PLcache+PL) incurs non-trivial performance overhead (26%). It is because locking introduces extra cache conflict misses over the protected cache lines. With larger caches and higher associativities, the number of conflict misses is reduced, resulting in smaller performance overhead (e.g. 15% of the baseline for 16KB direct mapped cache and almost 0% of the baseline for 16KB 2-way). Second, the RPcache has almost no performance difference compared to the baseline results. This is because the performance impact caused by random invalidation is small. These results also agree to that reported with the original PLcache and RPcache [23]. Third, the RPcache+IL has substantial performance overhead for 8KB direct mapped cache (63%). The reason is that because of frequent conflict misses, informing loads exception handler is invoked frequently, wasting lots of cycles. With larger caches and higher associativities, the cache is able to hold the working data set of AES and the RPcache+IL has low performance overhead (only 8% overhead for the 8KB 2-way cache and almost 0% overhead for the 8KB 4-way cache).

For our informing loads with the regular cache (Regular cache+IL), we also compare it to the software-based hybrid protection schemes (v1, v2 and v3 in Section 3.1). Since v1 and v3 always cause more than 6X slowdown to the baseline in our experiments, Figure 9 only includes the results for v2. From Figure 9, it can be seen that due to extra instructions (pointer indirection in our approach and S-box-based computation as well as permutation in v2), both v2 and Regular cache+IL have non-trivial performance overhead compared to the protection schemes with special cache designs. For 8KB direct mapped, 8KB 2-way, 16KB direct mapped and 32KB direct mapped caches, our scheme incurs higher performance overhead than v2. This is due to the large number of conflict cache misses over the protected cache lines, which lead to frequent invocation of the exception handler. As the cache size and set-associativity increase, such performance overhead quickly diminishes. The L1 data caches in modern commodity processors often have capacity bigger than 8KB and/or set associativity higher than one-way (as indicated in Intel® 64 and IA-32 Architectures Software Developer's Manuals). With those configurations, our approach incurs much smaller performance overhead compared to v2. For example, for the 16KB 2-way cache, our scheme incurs 40% overhead while v2 has 67%. For the 32KB 2-way cache, our scheme has 37% overhead and v2 has 67%. The main reason is that our proposed approach only responds to the potential dangerous event (i.e., cache miss over critical data), during which permutation updates and loading of all critical data are performed. In comparison, pure software approaches such as v2 have to perform these operations no matter whether there is a potential information leakage event or not. Furthermore, v2 offers relatively lower security levels because of its treatment of the inner rounds of AES as discussed in Section 3.1.

5.3.2 Performance impact on an SMT processor

In this experiment, we examine the performance impact of the protection schemes on SMT processors. We use two-way SMT processors and have AES run together with one of the ten SPEC2000 INT benchmarks (*bzip2*, *gap*, *gcc*, *gzip*, *mcf*, *parser*, *perl*, *twolf*, *vortex* and *vpr*). For each benchmark, SimPoint [20] is used to select a simulation phase of 300-million instructions. We vary the L1 data cache configurations from 8KB direct-mapped to 16KB, 32KB and 64KB 4-way set-associative. The results are shown in Figure 10. We report two metrics for each cache configuration, the overall instructions per cycle (IPC) for throughput and the Hmean metric (Harmonic mean of the IPC speedup/slowdown of each separate thread [9]) for fairness. Note that although here we use instructions per cycle as the performance metric for AES implementations, these IPCs are relatively normalized to the baseline's IPC in terms of AES throughput (cycles per byte) to ensure fair overall IPC comparison. Here, the baseline is the standard OpenSSL AES implementation and assume that it takes N instructions to encrypt a message. Due to added defense mechanisms, a software approach, e.g., v2, may require a different number of instructions (e.g., M) to encrypt the same message. To ensure fair comparison, the IPC of v2 is computed using the baseline instruction count, N. Our results exclude the instructions for the user-level exception handler.

From Figure 10, we can make the following observations on PLcache+PL and RPcache+IL. First, for the 8KB direct-mapped cache, the PLcache (and PLcache+PL) incurs performance degradation in throughput (8% on average) because locking introduces extra cache misses. When measuring Hmean, however, the PLcache reports 6% improvement. The reason is due to the memory-intensive benchmark *mcf*, which dominates the cache usage and significantly affects the performance of AES. With the PLcache, the AES lookup tables are locked, thereby reducing such negative impact from *mcf* upon AES. With *mcf* excluded, PLcache reports

a 4% loss on Hmean. The RPcache improves both the throughput (7%) and Hmean (7%) because the cache line relocation alleviates the cache conflict problem associated with the direct mapped cache. Second, for 16KB, 32KB and 64KB 4-way set-associative caches, the PLcache achieves very small throughput improvement. This is because those cache configurations have enough capacity for the working sets of both AES encryption and the SPEC CINT benchmarks and locking helps to avoid cache misses on the protected tables. The RPcache has a little degradation in IPC (1% for 16KB) because of the effect of invalidations from cache line relocation. This effect tends to diminish for large caches such as 32KB (almost 0%) caches due to their higher number of cache sets. These results also agree to that reported with the original PLcache and RPcache [23]. In terms of Hmean, the PLcache (PLcache+PL) and RPcache have very limited impact compared to the baseline caches. Third, the RPcache+IL incurs higher overhead compared to the original RPcache. The reason is due to user-level exception handling. For 8KB directed mapped and 16KB 4-way caches, the AES lookup tables are frequently replaced with the data from SPEC benchmarks. Therefore, the exception handler is invoked frequently, resulting in 49% and 45% losses in IPC and 19% and 57% losses in Hmean for those two cache configurations, respectively. The low Hmean results for the RPcache+IL are due to inherently unfair usage of informing loads. Since only the AES code uses the informing loads, the performance loss due to invocations of the exception handler is mainly upon AES rather than the SPEC workloads, thereby indeed being unfair. Such performance loss is recovered when the cache size is increased to 32KB (12% loss in IPC and 12% loss in Hmean) and 64KB (5% loss in IPC and 4% loss in Hmean) as the lookup tables will reside in the cache for much longer time.

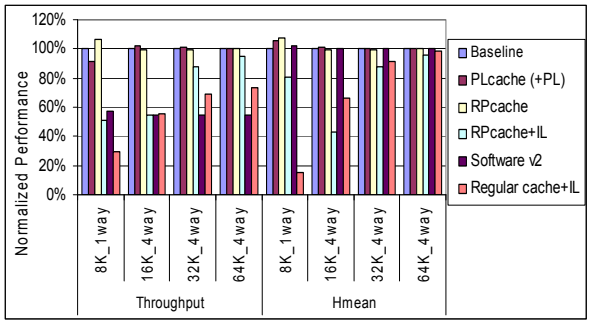

Fig. 10. Performance impacts of various protection schemes over various cache configurations in two-way SMT processors.

In Figure 10, we also show the performance on the most efficient software hybrid approach (Software v2) and the regular cache with informing loads scheme (Regular cache+IL) for two-way SMT processors. The results of v1 and v3 are not shown since both are much worse than v2. For throughput, both v2 and Regular cache+IL have significant throughput degradation for all cache configurations. In the meantime, Regular cache+IL gains higher throughput over v2 except for the 8KB direct-mapped cache. This is due to the diminishing number of conflict misses as the cache can hold the working sets of both threads. Therefore it results in a small number of invocations of the exception handler, which lead to small performance overhead. For fairness, v2 reports similar results to the baseline as v2 makes no use of informing loads, thereby is not affected. Regular cache + IL reports lower fairness results. The reason is due to the unfair usage of informing loads: informing loads are only used in AES but not in the other thread. Therefore, when an informing load exception happens, it only affects AES and does not affect the co-running thread. The fairness of Regular cache + IL improves quickly when the cache goes from 8KB up to 64KB as a result of the reducing number of invocations of the exception handler.

5.4 Summary

Protecting computer systems is often about selecting the tradeoff between security and efficiency. A stronger level of security protection usually comes at extra cost. Here we use performance overhead, hardware complexity, flexibility, compatibility with legacy programs and software change complexity as the factors for consideration and present a comparison among various protection schemes against software cache-based side channel attacks, as shown in Table 4. For security, although the PLcache and RPcache designs are effective against the two cache attacks analyzed in [23], they are still vulnerable to other cache attacks [14]. Our proposed hardware-software integrated approaches address the source of information leakage and are able to mitigate the latest cache attacks. Software hybrid approaches are also able to provide certain resistance against cache attacks but may trade security for performance, such as v2. From the perspective of performance overhead, the special hardware designs reduce the performance cost, while our informing loads with regular cache approach incurs non-trivial performance degradation because of extra pointer indirection. PLcache+PL and RPcache+IL pose a small to medium performance overhead upon the original cache designs, while pure software approaches usually incur the highest performance overhead. In terms of the ability to evolve when better software countermeasures are crafted or new attacks are developed, pure hardware designs lack the flexibility compared to software approaches, as evidenced by the vulnerabilities of the original RPcache designs to some timing-based attacks. Our proposed hardware-software integrated approaches combine some of the performance advantage of the hardware design with the flexibility of the software defense. Among them, the PLcache+PL and RPcache+IL provide performance efficiency at the cost of extra hardware and complexity and our approach for regular caches has smaller performance overhead compared to the pure software

Table 4. A comparison between various protection schemes

	PLcache	RPcache	PLcache+PL	RPcache+IL	Regular cache +IL	Software v1,v2,v3
Mitigation against access-driven attacks?	Yes, with SW preloading	Yes	Yes	Yes	Yes	Yes
Mitigation against time-driven attacks?	Yes, with SW preloading	No	Yes	Yes	Yes	Yes
Performance overhead	None-Small	None+	Small-Medium*	Small-Medium*	Medium-High*	High
Hardware changes	Trivial	Non-trivial	Non-trivial	Non-trivial	Light-weight	None
SW flexibility in handling misses	No	No	Yes	Yes	Yes	Yes
Compatibility with legacy programs	Yes with OS change	Yes	No	No	No	No
Software change complexity	Small	None	Medium	Medium	High	High

* Note: for small size and low-associativity caches, the performance overhead would be high
+Note: Performance can increase for RPcache for small cache size and direct-mapped caches

approach, for caches larger than 16 KB, while requiring only minor architectural support for informing loads. For compatibility with legacy programs, RPcache is able to provide protection without requiring any software changes. In terms of the complexity of software changes PLcache requires relatively small change in order to provide fine-grain locking control of cache lines and RPcache requires no software changes, while PLcache+PL and RPcache+IL need the user-level exception handler. In comparison, regular cache+IL and software v1, v2, v3 introduce software random permutation and other changes to the target programs.

6. Conclusions

Software cache-based side channel attacks pose serious threats to the security of computer systems. In this paper, we review current cache attacks and identify the weakness of existing hardware/software countermeasures. We then propose integrated hardware-software approaches to provide stronger security protection. We use preloading to protect the PLcache from the initial loading exposure. A light-weight hardware support, informing loads, is used to detect the sign of potential danger - cache misses of critical data and then to deploy flexible software countermeasures. We propose to use informing loads combined with simple yet effective software countermeasures to protect both the RPcache and regular caches. Experimental results show that our approaches achieve strong security protection at relatively low performance overheads.

Acknowledgements

We would like to thank our shepherd, Ruby Lee, and the anonymous reviewers for their valuable comments. This work was supported by an NSF CAREER award CCF-0747062.

References

[1] O. Acıiçmez. Yet Another MicroArchitectural Attack: Exploiting I-Cache. *ACM workshop on Computer Security Architecture (CSAW)*, 2007.

[2] O. Acıiçmez, Ç.K. Koç and J.-P. Seifert. On the Power of Simple Branch Prediction Analysis. *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007.

[3] O. Acıiçmez, Ç.K. Koç and J.-P. Seifert. Predicting Secret Keys via Branch Prediction. *The Cryptographers' Track at the RSA Conference (CT-RSA)*, 2007.

[4] D. Bernstein. Cache-timing attacks on AES. preprint, 2005, http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[5] J. Blommer and V. Krummel. Analysis of countermeasures against access driven cache attacks on AES. *Workshop on Selected Areas in Cryptography (SAC)*, 2007.

[6] J. Bonneau and I. Mironov. Cache-Collision Timing Attacks against AES. *Workshop on Cryptographic Hardware and Embedded Systems (CHES)* 2006. Source code available at http://www.jbonneau.com/research.html

[7] E. Brickell, G. Graunke, M. Neve and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *Cryptology ePrint Archive*, Report 2006/052, 2006.

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

[9] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically Controlled Resource Allocation in SMT Processors. *International Symposium on Microarchitecture (MICRO)*, 2004.

[10] J. Daemen and V. Rijmen. The design of Rijndael: AES - the advanced encryption standard. *Springer-Verlag*, 2002.

[11] S. Gueron. Advanced Encryption Standard (AES) Instructions Set. *White Paper, Intel Corporation*, July 2008.

[12] S. Gueron, J.-P. Seifert, G. Strongin, D. Chiou, R. Sendag and J. J. Yi: Where Does Security Stand? New Vulnerabilities vs. Trusted Computing. *IEEE Micro*, Nov.-Dec. 2007.

[13] M. Horowitz, M. Martonosi, T. Mowry and M. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. *International Symposium on Computer Architecture (ISCA)*, 1996.

[14] J. Kong, O. Acıiçmez, J.-P. Seifert and H. Zhou, Deconstructing New Cache Designs for Thwarting Software Cache-based Side Channel Attacks, *ACM Workshop on Computer Security Architecture (CSAW)*, 2008.

[15] M. Neve and J.-P. Seifert. Advances on Access-driven Cache Attacks on AES. *Workshop on Selected Areas in Cryptography (SAC)*, 2006.

[16] D.A. Osvik, A. Shamir and E. Tromer. Cache attacks and Countermeasures: the Case of AES. *The Cryptographers' Track at the RSA Conference (CT-RSA)*, 2006.

[17] D. Page. Defending Against Cache Based Side-Channel Attacks. *Information Security Technical Report*, volume 8(1): 30-44, 2003.

[18] D. Page. Partitioned Cache Architecture as a Side-Channel Defense Mechanism. *Cryptology ePrint Archive*, Report 2005/280, 2005.

[19] C. Percival. Cache Missing For Fun and Profit. *BSDCan* 2005. Available at: http://www.daemonology.net/papers/htt.pdf

[20] T. Sherwood, E. Perelman, G. Hamerly and B. Calder. Automatically Characterizing Large Scale Program Behavior. *International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[21] C. A. Thekkath and H. M. Levy. Hardware and Software Support for Efficient Exception Handling. *ASPLOS* 1994.

[22] K. Tiri, O. Acıiçmez, M. Neve, F. Andersen. An Analytical Model for Time-Driven Cache Attacks. *Fast Software Encryption workshop (FSE)*, 2007.

[23] Z. Wang and R. B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. *International Symposium on Computer Architecture (ISCA)*, 2007.

[24] OpenSSL: the open source toolkit for SSL/TLS. http://www.openssl.org/

[25] OpenSSL: Montgomery exponentiation side-channel local information disclosure vulnerability. http://www.securityfocus.com/bid/25163/, 2007.

[26] OpenSSL: implement fixed-window exponentiation to mitigate hyper-threading timing attacks, http://cvs.openssl.org/chngview?cn=13344, 2005.