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

Zhenghong Wang and Ruby B. Lee
Princeton Architecture Laboratory for Multimedia and Security (PALMS)
Department of Electrical Engineering, Princeton University, NJ 08544
{zhenghon, rblee}@princeton.edu

ABSTRACT

Software cache-based side channel attacks are a serious new class of threats for computers. Unlike physical side channel attacks that mostly target embedded cryptographic devices, cache-based side channel attacks can also undermine general purpose systems. The attacks are easy to perform, effective on most platforms, and do not require special instruments or excessive computation power. In recently demonstrated attacks on software implementations of ciphers like AES and RSA, the full key can be recovered by an unprivileged user program performing simple timing measurements based on cache misses.

We first analyze these attacks, identifying cache interference as the root cause of these attacks. We identify two basic mitigation approaches: the partition-based approach eliminates cache interference whereas the randomization-based approach randomizes cache interference so that zero information can be inferred. We present new security-aware cache designs, the Partition-Locked cache (PLcache) and Random Permutation cache (RPcache), analyze and prove their security, and evaluate their performance. Our results show that our new cache designs with built-in security can defend against cache-based side channel attacks in general – rather than only specific attacks on a given cryptographic algorithm – with very little performance degradation and hardware cost.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Miscellaneous;
K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security, Design, Performance

Keywords: Cache, Side channel, Computer architecture, Security, Processor, Timing attacks

This work was supported in part by DARPA and NSF Cybertrust CNS-0430487 and CNS-0636808, DoD and Intel.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

1. INTRODUCTION

Protecting the confidentiality of secret or sensitive information is a major concern for users of computer systems. This is often done by using cryptographic methods, so that even if the adversary gets hold of the data, it is encrypted and he cannot interpret it – unless he can get hold of, or discover, the key. Strong cryptography is designed so that it is computationally infeasible to infer the key bits by brute-force trials, or even by differential cryptanalysis [1] and linear cryptanalysis [2]. However, rather than use sophisticated mathematical analysis, side-channel attacks use auxiliary information to deduce key bits. They collect "side channel information", which can be in the form of timing, power consumption, radiation or sound produced by the system [3]. This often carries information about the cryptographic keys. For example, in a differential power analysis attack [4], a bit of secret key information can be discovered by detecting which branch is executed upon a key-dependent conditional branch, because the device does different things depending on which branch is actually executed, consuming different amounts of power.

Side channel attacks have mostly been used in attacking simple systems such as smart cards, due to the noisy nature of the side channel information, the difficulty in collecting such information and the need for physical access or proximity. In attacking more complicated general-purpose computer systems, more traditional attacks are used, e.g., exploiting flaws in operating systems that allow an attacker to gain direct access to the secrets or even subvert the entire system.

Unlike physical side channel attacks, *software* cache-based side channel attacks can impact a much wider spectrum of systems and users. This is because caches exist in almost all modern processors, the software attacks are very easy to perform, and are effective on various platforms [5-7]. This makes cache-based side channel attacks extremely attractive as a new weapon in the attacker's arsenal. Also, existing mitigation methods for side channel attacks are all *ad hoc* and only defend against specific attacks. No past work has proposed general solutions as we do in this paper. We propose cost-effective solutions that address cache-based side channel attacks in general, by eliminating the *root cause* of these attacks. Our main contributions are:

- An analysis of cache-based side channel attacks, identifying *cache interference* as the root cause that enables these attacks.

- Identification of two main mitigation approaches: the partition-based solutions and the randomization-based solutions.
- Detailed proposal of new security-aware cache architectures for each mitigation approach.
- Analysis of these new cache architectures and an information-theoretic proof of security.
- Performance evaluation of the proposed architectures using cycle-accurate simulation.

In section 2, we define our threat model and analyze different types of cache-based side channel attacks, showing how and why they work. In section 3, we discuss software solutions and hardware solutions that have been proposed, and their problems. In section 4, we propose two new general-purpose hardware solutions: the *Partition Locked cache (PLcache)* and the *Random Permutation cache (RPcache)*. In section 5, we evaluate the proposed designs in terms of security and performance. In section 6 we review past work, and we conclude in section 7.

2. THREAT MODEL AND ATTACKS

2.1. Threat Model and Assumptions

The goal of the adversary is to learn information that he has no legitimate access to, e.g., the classified data or secret keys. The attacker needs very little capability to mount a cache side channel attack. An adversary is one or multiple unprivileged user processes, including a remote client that can interact with the server where the secrets are stored. The adversary has no administrator privilege. We assume that the adversary does not exploit physical attacks like bus and memory probing, since he typically does not have physical access to the victim machine and such ability is not necessary for cache-based side channel attacks. The adversary can achieve his goal without the need for finding and exploiting system flaws, but rather just acts like a normal process, performing legitimate operations. We further assume that the victim and the adversary are "isolated" processes that do not share the same address space, since this always gives the adversary the ability to infer information about the victim's behavior.

Many different types of cache side channel attacks are possible. We group them based on the attacker's ability to observe cache accesses of the victim process. The attacker may be able to directly detect each individual access of the victim. Alternatively, he may only be able to take a snapshot of the cache in a certain time period and see several accesses without knowing their order. Sometimes, he cannot observe any cache access, and can only measure the overall execution time of the victim process.

2.2. Percival's Attack on RSA

Modern microprocessors, such as Simultaneous Multi-Threading (SMT) processors, allow multiple threads to run simultaneously, sharing part of the cache subsystem. This gives an attacker process the ability to *directly observe* other concurrent threads' cache accesses and obtain a relatively accurate trace. In 2005, Percival [6] demonstrated an attack against the popular OpenSSL implementation of the RSA algorithm using this approach.

Attack description: The attacker manages to run simultaneously with the victim process which is performing RSA encryption. His goal is to discover bits of the private encryption key used by the victim. The attacker sequentially and repeatedly accesses an array, thus loading in his own data to occupy all cache lines; at the same time he measures the delay for each access to detect cache misses, e.g., using the `rdtsc` instructions to read a timer in Intel x86 processors. The victim's cache accesses will evict the attacker's data, causing the attacker to miss on these cache lines, enabling detection by the attacker.

Attack Analysis: The core operation used in RSA is modulo exponentiation. It is often implemented with a series of squarings and multiplications. The encryption key is also divided into a series of segments. For each multiplication, a multiplier is selected from a set of pre-computed constants stored in a table. During the table lookup, a segment of the encryption key is used to index the table. As the table is stored in memory, the attacker can detect the cache evictions caused by the victim (the encrypting process) for the table lookup. Based on which line is evicted, the attacker can infer which table entry is accessed. This tells the attacker the index used for this table lookup, which is a segment of the encryption key.

2.3. Bernstein's Attack on AES

In Bernstein's attack [5], the attacker has no direct observation of the victim process' cache accesses. He may be on another machine, performing the attack remotely. He can only observe the total execution time of a program.

Attack description: The victim is a software module that can perform AES encryption. The module is a "black box"; the user is only able to choose the input to the AES software module and measure how long it takes to complete the encryption. The user may be a process in the same machine or a remote user requesting encryption service. Empirical studies show that for most software AES implementations running on modern microprocessors, the execution time of an encryption is input-dependent and can be exploited to recover the secret encryption key. The attack consists of three steps:

1. *Learning phase*: Let the victim use a known key $\boldsymbol{K}$. The attacker generates a large number, N, of random plaintexts $\boldsymbol{P}$. He sends the plaintexts to the cipher program and records the encryption time for each plaintext. He uses the algorithm shown in Figure 1 to obtain the timing characteristics for $\boldsymbol{K}$, shown in Figure 2(a).
2. *Attacking phase*: Repeat step 1 except that an unknown key $\boldsymbol{K'}$ is used. The timing characteristics for $\boldsymbol{K'}$ is shown in Figure 2(b). Note that the input set is randomly generated and not necessarily the same as that used in step1.

```
For key K:
For s = 1 to N do begin
    Generate a random 128-bit Plaintext block, P_s;
    T_s = time taken for AES encryption of P_s using K;
end;
For i = 0 to 15 do begin
    For j = 0 to 255 do begin
        count = 0;
        For s = 1 to N do begin
            If p_i = j then
                TSUM_i(j) = TSUM_i(j) + T_s;
                count = count+1;
        end;
        t_avg^i(j,K) = TSUM_i(j)/count;
    end;
end;
```

(a)

```
For i = 0 to 15 do begin
  For j = 0 to 255 do begin
    Corr[j] = Σ_{m=0}^{255} [ t^i_avg(m,K) • t^i_avg(m ⊕ j, K') ]
  end;
  k_i' = findMax(Corr);
end;

Note: Function findMax() searches for the
maximum value in the input array and returns
its index.
```

(b)

Figure 1. (a) Timing characteristic generation (b) Key-byte searching algorithm

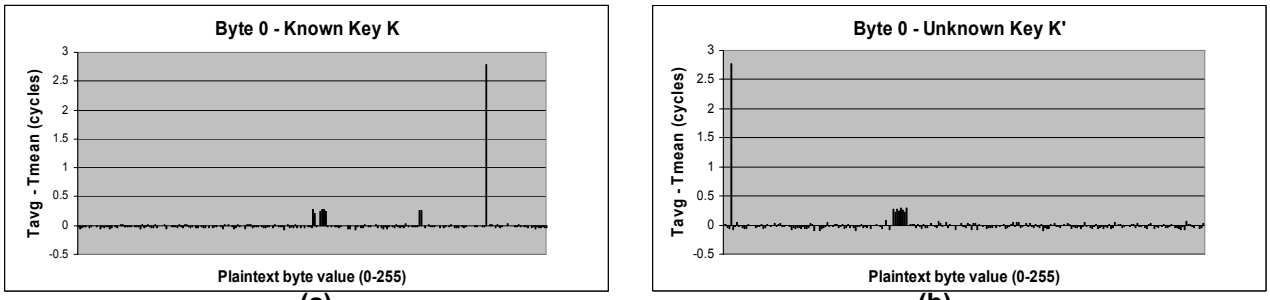


Figure 2. Timing characteristic charts for byte 0 (obtained on a Pentium-M machine)

3. *Key recovery*: Given the two sets of timing characteristics, use the correlation algorithm shown in Figure 1(b) to recover the unknown key ***K'***.

In Figure 2, the height of the bar at position j is $t_{avg}^{\ i}(j,\boldsymbol{K})$, which is the average of the execution time of the AES encryptions when the i-th byte of plaintext ***P*** is j, using key ***K***. In the AES algorithm, each plaintext ***P*** is an M-byte block, e.g., M=16, therefore M pairs of such timing characteristic charts are generated. Figure 2 only shows one such pair, corresponding to byte 0 in ***P***. Experiments show that $t_{avg}^{\ i}(j,\boldsymbol{K})$ is pretty much fixed for a given system configuration. Furthermore, it is found that when a different key ***K'*** is used, the timing charts roughly remain the same except that the locations of the bars in the charts are permuted, as shown in Figure 2. More specifically, the following equation holds:

$$t_{avg}^{\ i}(p_i, \boldsymbol{K}) = t_{avg}^{\ i}(p'_i, \boldsymbol{K'}) \text{ if } p'_i \oplus k'_i = p_i \oplus k_i \qquad (1)$$

where $\oplus$ is the bit-wise XOR operation, and k_i and k'_i are the i-th byte of ***K*** and ***K'*** respectively.

Attack Analysis: Table lookups are intensively used in various AES implementations for high performance. For example, OpenSSL v0.9.7a uses five tables. During the encryption, for each byte p_i of the plaintext, one table is accessed using the index $(p_i \oplus k_i)$ where k_i is the i-th byte of the encryption key. Ideally, these table lookups will hit in the cache since normally the cache is large enough to accommodate all these tables. However, in reality it is found that there are always other memory accesses that regularly contend for cache lines at some fixed locations. Therefore, given an index $(p_i \oplus k_i)$, if the corresponding table entry is mapped into one of these "hot" cache locations, the table lookup will experience a cache miss, and will lead to larger $t_{avg}^{\ i}(p_i, \boldsymbol{K})$, i.e., a high bar in Figure 2. Also, when $p'_i \oplus k'_i = p_i \oplus k_i$ the table lookup will access the same table entry, i.e., the same cache location. This explains why the same bar in Figure 2(a) also appears in Figure 2(b), though at different location, as described by equation (1).

The two representative attacks analyzed above correspond to two extremes in the attacker's ability to observe cache interference. There are also other attacks reported [8], where the attacker has an observation ability between these extremes. Despite the dramatic difference in these attacks, they all rely, directly or indirectly, on *cache interference*. In attacks like Percival's attack, *external interference* is exploited by a process outside the victim program. The victim's cache accesses evict the attacker's cache lines and therefore can be observed. In Bernstein's attack, *internal interference*, coming from the victim code module itself, occurs. The cache line evictions of AES table entries are caused by another part of the software module (e.g., the wrapper code for the AES encryption core) and even the encryption code itself. These two types of cache interference are not a result of any specific cache architectures. They are rather general and almost all microprocessors with caches are vulnerable to such attacks.

3. EXISTING SOLUTIONS

3.1. Software Solutions

Most existing solutions for cache-based side channel attacks are software based and specific to a given encryption algorithm. The basic methodology is to rewrite the software in a way that the known attacks can not succeed. For example, to mitigate the attacks against AES, several new implementations have been proposed, e.g. 1) pre-load the AES tables into the cache before starting an encryption so that all accesses to AES tables hit in the cache and hence have constant encryption time; or 2) do not use table lookups at all in the AES implementation – use only mathematical operations instead. More proposals can be found in [7][9]. To mitigate attacks against RSA, one proposal changes the pre-computed multiplier table such that to access any multiplier, all cache lines in the table are touched. The attacker always observes a fixed cache access pattern and can not guess the key bits.

One problem with these software solutions is that they are all *ad hoc* and attack specific. They are tailored to a given program (which has to be changed) and only mitigate the known attacks. New attacks are still possible as the underlying cache interference still exists. A second problem is that the software solutions often cause significant performance degradation. The new software implementations of AES are 2X to 4X slower than the original insecure implementation [9]. A third problem is that some software solutions rely on specific hardware architecture parameters. For example, the new RSA implementation above needs to be rewritten if the cache line size changes. This is undesirable for software portability. Finally, some software countermeasures have been proved not sufficient. For example, pre-loading AES tables before encryption indeed can not ensure constant encryption time. The table entries can still be evicted after the tables are loaded and before or during the encryption.

3.2. Hardware Solutions

Hardware solutions proposed include conceptual ones [7] which disable the cache or use separate caches for simultaneous threads. Some new eviction strategies which minimize the extent to which one thread can evict data used by another thread were suggested in [6]. In [18], Page proposed to exploit a partitioned cache originally designed for multimedia applications to block cache-based side channel attacks. The ISA is changed to make the cache a visible part of the architecture, with new instructions that can define a partition and specify its size and other parameters, the cache line size, the stride size, etc. However, the author also admitted that the cost of the design and its performance impact can be high.

4. NEW HARDWARE SOLUTIONS

Unlike the software solutions that are *ad hoc* and attack specific, our work attempts to eliminate the root cause of the problem for cache side channel attacks in general. We also aim at designs that can leverage existing cache features as much as possible, introducing low cost changes only when necessary. We feel this is necessary to encourage rapid and widespread deployment. Our results show that with little hardware cost, this goal can be achieved without impacting performance. Section 2 showed that cache interference is the root cause for cache-based side channel attacks. To block these attacks, we can try to eliminate cache interference, i.e., prevent inference of cache line evictions. We identify two main solution approaches. One class of solutions essentially partitions the cache, so that there is no sharing of cache lines, and hence no interference. The other approach allows sharing, but randomizes the cache interference, so that no useful information can be deduced. We describe an efficient hardware solution for each approach.

4.1. Partition-Locked Cache (PLcache)

The concept of cache partitioning is not new, as described in section 3. However, in previous designs, the partitions are mostly static. This prevents sharing, often leading to large performance degradation. A process may use very few cache lines in its partition, but unused lines are not available to other processes which may need more cache lines than they have in their partitions. We refer to such a cache as a statically partitioned cache, or a *partitioned cache* in short. In this paper, we propose the Partition-Locked cache (PLcache) that essentially achieves the effect of cache partitioning, but much more flexibly with less performance degradation. In PLcache, the cache lines of interest are locked in cache, creating a flexible "private partition"; these cache lines can not be evicted by other cache accesses not belonging to this private partition, preventing internal, as well as external, cache interference.

4.1.1. Architecture Description

The PLcache consists of two parts: the hardware addition to the cache and the system interface for controlling which cache lines should be locked.

A. Hardware addition: Figure 3 shows the hardware addition to the cache, consisting of two new tags, L and ID, per cache line. The 1-bit L flag indicates whether this cache line is locked or not. The ID field indicates the owner of the cache line. Not shown in Figure 3, is an optional LL bit per TLB entry, page-table entry or segment descriptor (if the architecture supports segmentation) which indicates if an access to a page or a segment should cause the corresponding cache line to be locked in cache.

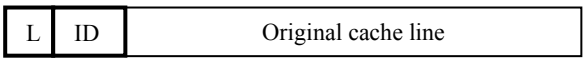


Figure 3. A cache line of the PLcache

B. Control Interface: There are two mechanisms that allow the programmer, compiler and OS to control what to lock in the cache. Either mechanism can be implemented:

Table 1: ISA extension for PLcache

Name	Description
ld.lock/ ld.unlock	Identical to a normal load instruction with the additional action: If the memory access hits in the cache or causes a cache line to be fetched into the cache, the L bit of the cache line is set/cleared.
st.lock/ st.unlock	Identical to a normal store instruction with the additional action: If the memory access hits in the cache or causes a cache line to be fetched into the cache, the L bit of the cache line is set/cleared.

Table 2: API calls for PLcache

Declaration
int lock_mem_region (unsigned long start_addr, unsigned long length);
int unlock_mem_region(int region_id);

1) ***ISA extension***: a new set of load/store instructions with a lock/unlock sub-op can be added to the base ISA (Instruction Set Architecture). This gives the programmer or compiler the fine-grain control on what data to lock. Table 1 describes the new load/store instructions.
2) ***Segment/Page-based protection***: Regions of memory, e.g., those containing AES and RSA tables, can be marked as LOCKED. Accesses to such regions of memory should cause the corresponding cache line to be locked. This uses the LL bit described above, added to the segment descriptor and the TLB entry. This interface gives the operating system an opportunity to control what data should be locked in the cache. Table 2 shows API calls that can be exposed to programmers to make use of this mechanism. To lock a memory region, the function lock_mem_region() can be called which returns a region id. The LL bit of the corresponding segment is set. To unlock a region, the function unlock_mem_region() can be called with the id of the region to be unlocked as the input argument. The LL bit of the corresponding segment is cleared, and the locked cache lines invalidated.

C. Cache access handling: Figure 4 shows the flow chart of an access to a PLcache. Note that the sequential steps shown in the flow chart do not necessarily execute sequentially in the hardware. The *cache hit* handling procedure is the same as in traditional caches except that the L bit of the cache line accessed needs to be updated if necessary. If the access is a load/store instruction with lock/unlock sub-op, the instruction itself determines if the L bit should be set or cleared. This information is available early in the pipeline (after the instruction decoding stage) and hence does not impact cache access time. If the LL bit in segment descriptors is implemented, its checking can be done together with the checking of existing protection bits, and no extra delay is added. Similarly, if the LL bit in the TLB entry is implemented, the check can be done together with that for existing protection bits during the TLB access.

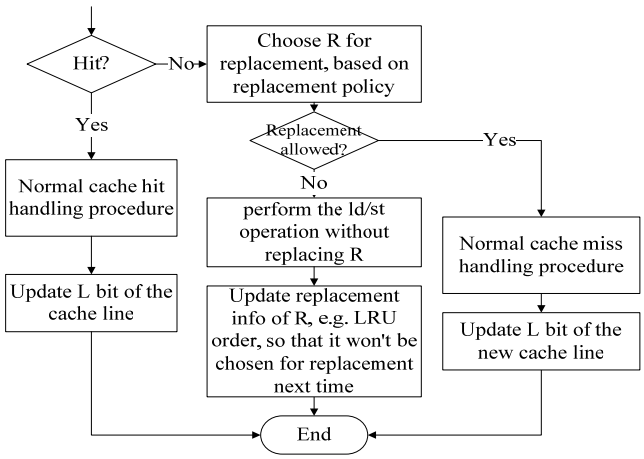


Figure 4. Access handling procedure for PLcache

During a *cache miss*, the replacement algorithm differs from a traditional cache because of the Locked cache lines. Let R denote the line chosen to be evicted by the normal cache replacement algorithm (e.g., LRU) and D denote the new data block that is being fetched into the cache. The following cases need to be considered:

Case	Description
1	If D does not need to be locked and R is also not locked, D replaces R like in a normal cache miss.
2	If D does not need to be locked but R is a locked line, D can not replace R. In this case, for a load instruction, one can simply return D to the processor execution core. For a store instruction, the data is written back to the next level of memory, without replacing R. The LRU list should be updated so that R becomes the most recently used line and will not be chosen for eviction next time. This can avoid repeatedly missing on this cache set due to the locked line.
3	If D needs to be locked in the cache, it is allowed to replace any line that is not locked or any locked line that belongs to the same process. We do not allow the new line to evict a locked line of another process. Such a miss can be handled as described in case 2.

D. Updating the L bit of a cache line: If the ISA extension is implemented, the instructions with locking/unlocking capability can set or clear the bits whereas normal load and store instructions can not. If the segment/page based protection is implemented, in each memory access the address is checked and the L bit is set or cleared accordingly. If both mechanisms are implemented, locking/unlocking instructions always set/clear the L bit, and a normal load/store instruction can also set the L bit if the address is in a locked memory region.

4.1.2. Discussion

ISA extension vs. segment/page-based protection: The ISA extension gives the software developer the flexibility to prevent cache interference for any portion of its memory. Legacy code however can not benefit without modification. The segment/page based protection provides a rather coarse-grain control mechanism – but both future code and

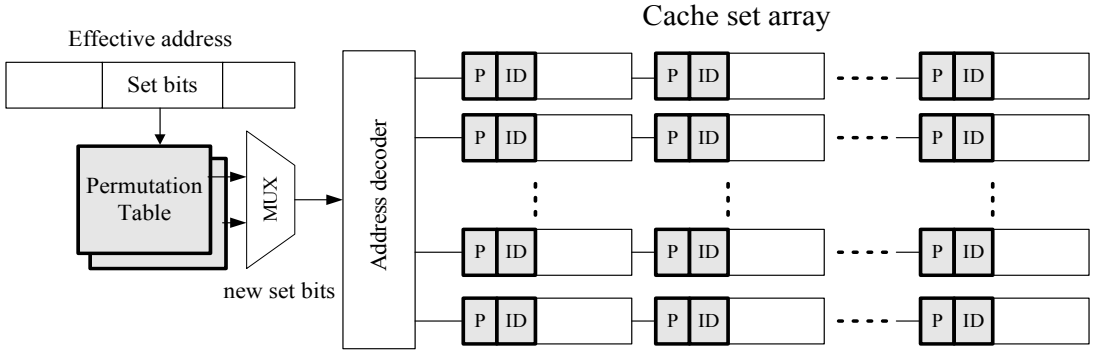


Figure 5. A logical view of the RPcache

legacy code can benefit from it. For example, the programmer can exploit the API calls to specify a memory region to be protected, and the OS can mark memory regions such as AES or RSA tables used by crypto libraries during load time.

Controlling the use of locking mechanisms: The proper use of PLcache will not allow any program to lock cache lines without OS oversight. Otherwise, a process may, maliciously or naively, lock excessive amounts of data in the cache, causing a security or fairness problem, respectively. An adversary can also selectively lock certain lines to interfere with other processes. In PLcache, the hardware only provides the locking mechanisms, and the software should ensure their proper use. For example, the programmer and compiler can specify and optimize what to lock, and the OS determines if the lock is allowed based on the security policy. This might allow only trusted processes to lock cache lines and might impose an upper bound on the number of cache lines that a process can lock. For our segment/page-based PLcache mechanism, the OS can make this decision during the API call for locking a memory region, denying this service when necessary. For our ISA-based PLcache mechanism, the OS can disable the locking instructions, e.g., treating them as normal memory instructions without locking capability. This can be done, via a per-thread "disable locking" flag which is used to guard the L-bit update logic in the pipeline.

Cache line ID management: Any hardware implemented field has a limit on the number of items that it can represent. Hence, an n-bit ID field of a cache line limits the maximum number of processes that can own lines in the cache *at any one time* to 2^n. This does not limit the total number of concurrent software processes that the OS can support. For example, non-critical processes that do not need to be isolated can share the same ID value, e.g., '0'.

4.2. Random Permutation Cache (RPcache)

We propose a Random Permutation Cache (RPcache) for the randomization-based approach. In contrast to the PLcache, this approach allows cache sharing, but randomizes the resulting interference, so that no useful information about which cache line was evicted can be inferred.

An attacker can observe another process's cache access only if that process changes the attacker's cache usage, i.e., evicts the attacker's cache lines. If the process evicts its own cache lines, the attacker has no way to know that. As shown in section 2, by knowing which cache lines have been accessed by the victim process, the attacker can infer critical information about the victim process. In RPcache, each time such cache interference occurs, we randomize it such that the interference carries no useful information.

Architecture Description: We assume a generic set-associative cache where M bits of the effective address, the *set* bits, are used to index the cache set array. The number of cache sets in the array is 2^M and each cache set contains N cache lines for an N-way set-associative cache, including direct-mapped caches where N=1.

A. Permutation of memory-to-cache mapping

A key operation that the RPcache performs is the permutation of the memory-to-cache mapping. Conceptually, this is done by using a level of indirection in indexing the cache. In RPcache, the memory-to-cache mapping for a process is stored in a *permutation table* (PT), as shown in Figure 5. The table has the same number of entries as the number of cache sets, and each entry contains a different M-bit number which indicates the new set. For each cache access, the PT is indexed with the M set bits of the effective address to obtain the new set bits, which are then used to index the cache set array. A complete randomization of the memory-to-cache mapping can be achieved by a random permutation of the contents of the table entries. This can be decomposed into a series of swap operations, each of which exchanges the contents of two entries. Swapping the *k*-th and the *i*-th table entries means changing the memory-to-cache mapping, $k \rightarrow S$ and $i \rightarrow S'$, to the new mapping $k \rightarrow S'$ and $i \rightarrow S$. This indirect indexing scheme is a logical description. In hardware, this extra level of indirection is not necessary, as we show later.

In the RPcache, a number of permutation tables are added and each table can be used by one or more processes to access the cache. For example, an encrypting process can use one table and all other non-critical processes use another. The number of such tables implemented depends on

needs and cost. In PC systems where only occasionally a process needs to be protected, one table should be enough. All other processes can use the original mapping that does not need a remapping table. The memory-to-cache mapping needs to be updated from time to time, during the execution of the process, as described later. Similar to PLcache, a P bit and ID field are also added to each cache line.

B. Randomization of cache interference

We first define terms we will use in our discussion.

Name	Description
R , S	R is the cache line being replaced in cache set S.
R' , S'	R' is the cache line being replaced in another cache set S' which is randomly selected.
D	The memory block being fetched into the cache.
P_X	The P-bit of cache line X, e.g., of R, R' or D.

In the case of cache interference between the victim and attacker processes (external interference), the interference occurs only when the victim evicts a line of the attacker. In RPcache, rather then replacing line R, another cache set S' is randomly selected with equal probability. The new line D that is to be put into the cache then replaces R' in S' instead of R in S. The memory-to-cache mappings of S and S' are swapped such that next time when the victim process wishes to access D, he will access the correct cache line. From the attacker's point of view, when he detects a cache miss, the cache miss can be caused by the victim's access to any cache set, with equal probability. Hence he can learn nothing about the address that the victim accessed. Note that after swapping the memory-to-cache mapping of S and S', if the process wishes to access another cache line originally in set S, it will now access set S'. It will miss on set S' and bring another copy of the line into set S' although set S still has it. To avoid this, ideally the cache lines in S and S' that belong to the current process should also be swapped. However, for efficiency we invalidate all such lines in S and S' and write them back to memory if they are dirty. Future accesses to them will get them correctly from the next level of the memory hierarchy. Since the selection of S' is independent of S, R and D, it can be pre-computed and the write-backs can be performed in the background to hide the associated overhead.

In the case of cache interference from other code in the victim's own process (internal interference), a similar idea can be applied. To distinguish the memory region to be protected from such internal interference, two fields, a P bit and ID field are added to each cache line (shown in Figure 5), similar to the L bit and ID field in the PLcache. An internal cache interference occurs if the new line D is non-protected while the old line R is protected, or if D is protected and R is non-protected. As the attacker can not directly observe internal cache interference (since the evicted lines belong to the victim himself), the attacker can only observe the overall effect like the encryption time in Bernstein's attack. If such internal interference is rather fixed, or repeatable, like the eviction of AES table entries at fixed locations, the attacker can learn the fixed interference by performing a large number of trials, observing the cipher's execution time for each trial, and using statistical analysis of these times. Therefore by randomizing every internal cache interference there will not be any repeatable interference (which carries information) that can be observed by the attacker. To randomize internal cache interference, each time when the new line D and the old line R have different P-bit values, R is not replaced. D is returned to the execution core if it is a load, or written to the next level of the memory hierarchy if it is a store, without replacing any line in the cache. At the same time, a cache set S' is randomly selected, and a line R' in S' is evicted. Then the original cache interference on R is now on R' which is purely random and not repeatable.

The mechanisms for controlling which cache lines should be protected are similar to those used in the PLcache except that no new instructions are needed. In addition to the P bit and ID field in each cache line, a PP bit is also added to segment descriptors or the TLB entries. By using the segment/page based protection mechanism described for the PLcache, the OS and programmer can specify the memory region to be protected. In addition, if a section of code is marked as protected, i.e., the code segment descrip-

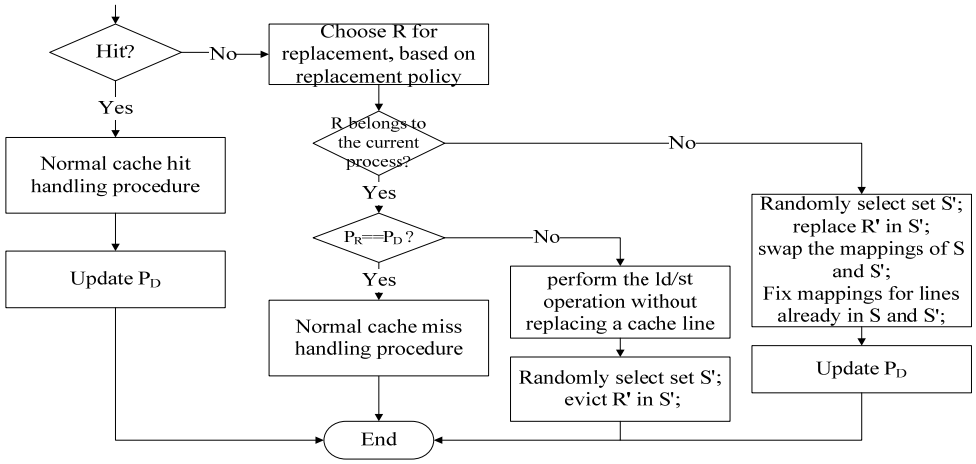


Figure 6. Cache access handling procedure for RPcache

tor or the ITLB entry has its PP bit set, any cache accesses issued by the protected code will set the P bit of the touched cache lines. This gives a convenient way for the OS to protect critical modules, e.g., the crypto libraries. The OS only needs to set the PP bit of the code pages of such modules.

C. Cache access handling

Figure 6 shows the flow chart of the cache access handling procedure. A cache hit in the RPcache is the same as a normal cache hit except that the P-bit of the cache line needs to be updated, based on the value of the PP-bit. During a cache miss, a line R in set S is chosen using the normal cache replacement policy. If R belongs to another process, a random set S' is selected. The new line D then replaces R' in S' and the memory-to-cache mapping for S' and S is swapped. If R belongs to the same process, two cases need to be considered, as shown below.

Case	Description
1	If $P_D == P_R$, R is replaced by the new line like in a normal cache miss.
2	If $P_D != P_R$, R can not be replaced and the access is performed without replacing R. R's replacement information is updated so that it will not be selected for eviction next time. This avoids repeated misses in set S. At the same time S' is randomly selected with equal probability among all cache sets, and R' in S' is evicted, based on the normal cache replacement policy for blocks in a set.

Low-overhead RPcache Implementation: Using an extra level of indirection in cache indexing can introduce extra delay into the cache access. For an L2 or L3 cache, a straightforward table lookup implementation may be good enough since one extra cycle in L2 or L3 cache loads will not cause much performance loss. However, for an L1 cache, which is often the most delay-sensitive module in a processor, an extra cycle on a cache hit may be unacceptable. We now show that indirect indexing for our RPcache can be implemented, without requiring an extra cycle, nor extending the cycle time latency.

Figure 7 shows the modified decoder circuitry for the RPcache based on the common implementation with the 3-to-8 NAND pre-decoder and the second stage NOR gates. Rather than having a fixed connection for each input of the NOR gate with one output of a 3-to-8 NAND pre-decoder, each input line of the NOR gate is connected via switches to all of the 8 output lines of the pre-decoder. The switches are controlled by a register called the *permutation register(PR)*, and at any time only one switch is on. Each permutation register is one entry of the permutation table in Figure 5. Note that we omit the MUX in Figure 5 for clarity. Compared with the original decoder, the only extra delay in the critical path is caused by the switch transistor. The path from the PR to the output of the NOR gate is not the critical path since the PR can be read out early in the pipeline instead of at the beginning of the cache access cycle: once the instruction is known as a memory-accessing instruction and to which process it belongs, the PRs can be read out and properly selected by the MUX. The delay caused by the switches is mainly due to the drain capacitance of the switch transistors which increase the load capacitance of the 3-to-8 NAND pre-decoders. To overcome this, we implement multiple copies of the pre-decoders, and let each of them drive a portion of the vertical lines such that the load of each NAND gate does not increase much. We also manually adjust the transistor sizes along the critical path, including the address bit drivers, the NAND gates, and the switches. We also insert a buffer between the address bit driver and the pre-decoders. We model this using cacti-3.2 tool [10], assuming a 0.18um technology. Table 3 shows the simulated results, where we first optimized the access time to less than 5% increase, then optimized the power to less than 10% increase. The increase in percent is relative to the unmodified cache modeled in cacti-3.2. Our results show that we can achieve approximately the same cache access time with up to 10% increase in power consumption. This is a straight forward implementation and further circuit optimization can certainly lead to even better designs.

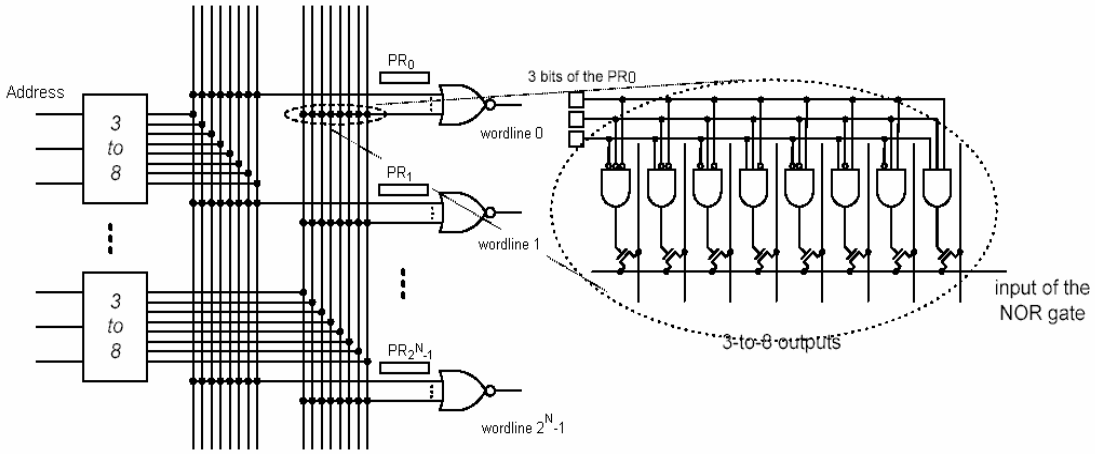


Figure 7. Address decoder circuitry of the RPcache

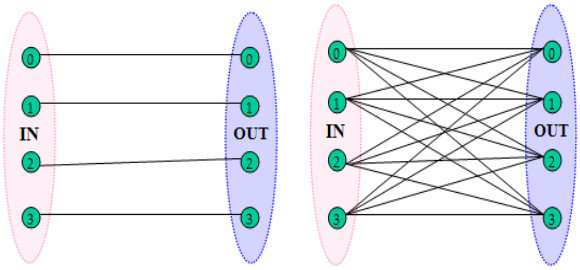
Table 3. Timing and Power Estimation of RPcache

RPcache	16K 2way	32K 2way	16K 4way	32K 4way
Access time(ns)	1.225 (+2.1%)	1.331 (+1.7%)	1.293 (+1.1%)	1.344 (+3.3%)
Power (nj)	1.205 (+8.6%)	1.282 (+1.3%)	1.792 (+6.1%)	1.906 (+2.1%)

5. Evaluation

5.1. Security Analysis

Security analysis of the PLcache: In a PL cache, the critical cache lines of the victim are locked in the cache, and the victim's accesses to these lines will always hit in the cache without causing any evictions of the attacker's cache lines. The attacker therefore can not learn anything about the victim's accesses to these lines. This defeats the Percival-type attack. Similarly, accesses from other parts of the code in the same process also can not interfere with the accesses to the critical cache lines.



(a) In traditional cache (b) In RPcache

Figure 8. Channel model of the cache-address-based side channel

Security analysis of the RPcache: We model the cache side channel as a communication channel (Figure 8), and prove that in the RPcache this side channel has zero channel capacity, meaning no information can be inferred by the attacker based on his observation of cache misses.

In the case of cache interference between processes, the victim and the attacker processes are the sender and the receiver in the channel, respectively. In the case of internal cache interference, either the protected code or the unprotected code in the same process can be the sender or the receiver (since they mutually interfere with each other). The *input alphabet* of the channel is the cache set number that the sender has accessed. The *output alphabet* is the cache set number where the receiver observes a miss. Both input and output alphabets are 2^M in size. We model the channel as a noiseless discrete time synchronous channel, where every access of the sender has an outcome at the output of the channel (i.e., evicts a cache line that belongs to the receiver) and can be observed by the receiver without error. This is the ideal case of the real cache side channel, and its capacity is the upper bound of the real channel. Figure 8(a) is the channel model for the traditional cache, which has a capacity of $\log_2(2^M)$=M bits per channel use [11], where 2^M is the total number of cache sets. Figure 8(b) is the channel model for the RPcache. In the RPcache, each time the sender evicts one of the receiver's cache lines, the receiver will experience a miss. However, as explained in section 4.2, this miss can be caused by the sender's access to any cache set, with equal probability. In other words, given an output symbol j, the probability that it is caused by an input symbol i is equal for any i. We then have the following theorem.

Theorem 1: In an RPcache, the capacity of the side channel based on cache line addresses is zero.

Proof:

Let $Pr(j|i)$ denote the conditional probability that given the input symbol i, the output symbol j is observed:

$Pr(j|i)$ = Prob(output = j | input = i)

The set of such conditional probabilities is called the channel matrix, which determines the channel capacity. According to section 4.2(B), the following relation holds:

$Pr(j|i) = Pr(j'|i')$ for any i,j and i',j'

In information theory, it is straight forward to prove that a channel with such a channel matrix has a zero capacity [11]. □

In Percival's attack, the attacker can detect cache misses caused by the victim's accesses. But according to Theorem 1, the attacker can learn nothing about the victim, and hence the attack can not succeed.

In Bernstein's attack, the victim is the AES code within a module. The attacker can not directly observe the output of the channel and can only see an aggregate version of the outputs, e.g., the execution time of the overall program. As we discussed in section 4.2(B), RPcache makes the interference to the AES table completely random. Therefore the attacker will not be able to generate the timing characteristic charts shown in Figure 2. The average time $t_{avg}^{\ i}(p_i, \boldsymbol{K})$ will be about equal for all i, p_i and $\boldsymbol{K}$. No key information can be inferred from the correlation between the two charts in Figure 2.

5.2. Performance Evaluation

We implemented the PLcache and the RPcache on M-Sim v2.0 [12] which is a multi-threaded microarchitectural simulation environment based on simplescalar3.0d. AES is used to evaluate the performance impact of the new cache architectures on code being protected. The SPEC2000 benchmark suite is used for evaluating the performance impact on general purpose workloads. In SPEC2000 benchmark simulation, the appropriate number of instructions are fast forwarded, ranging from 100 million to 2.1 billion instructions. Cycle-accurate simulations are then performed for 100 million instructions. Table 4 shows the simulation parameters used.

Performance impact on the protected code: Figure 9 shows the performance of the OpenSSL 0.9.7a implementation of AES on a processor with a traditional cache (Baseline), an L1 PLcache and an L1 RPcache. A total of 5 Kbytes of data need to be protected in this AES implementation. The

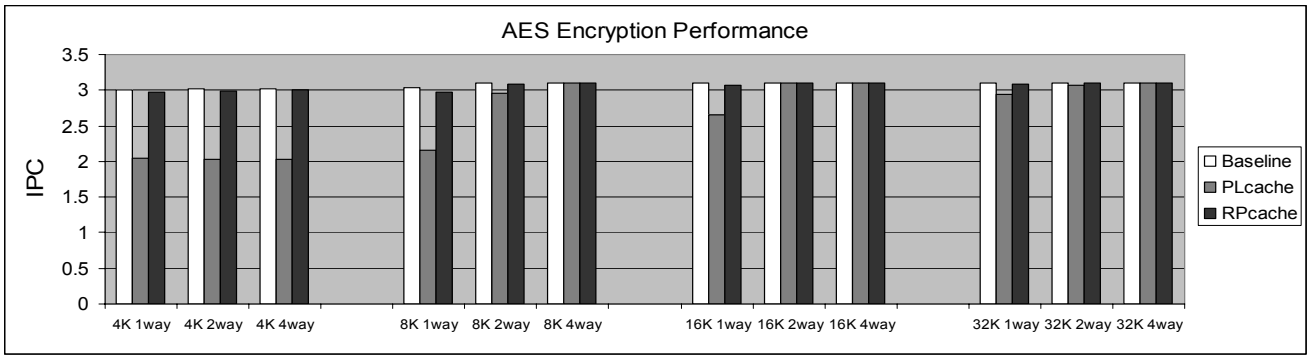
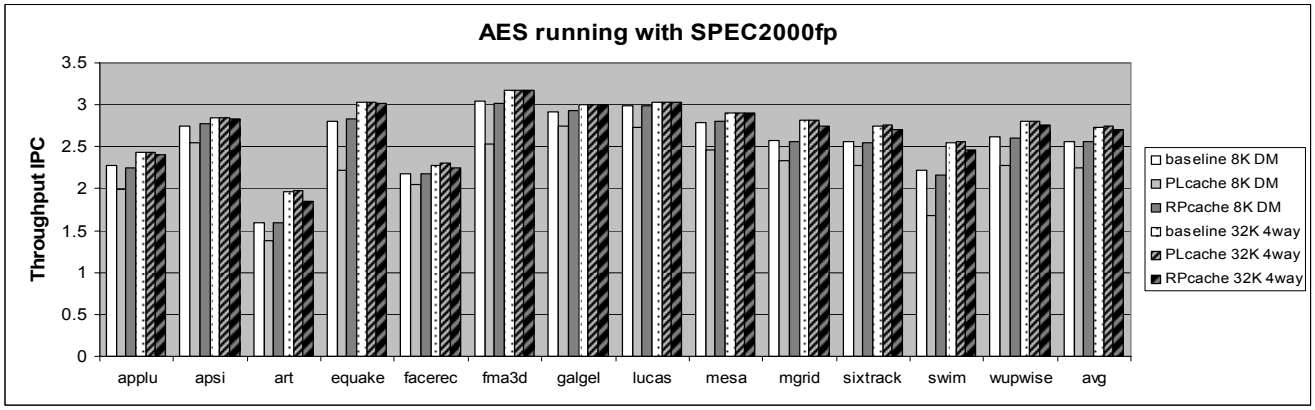
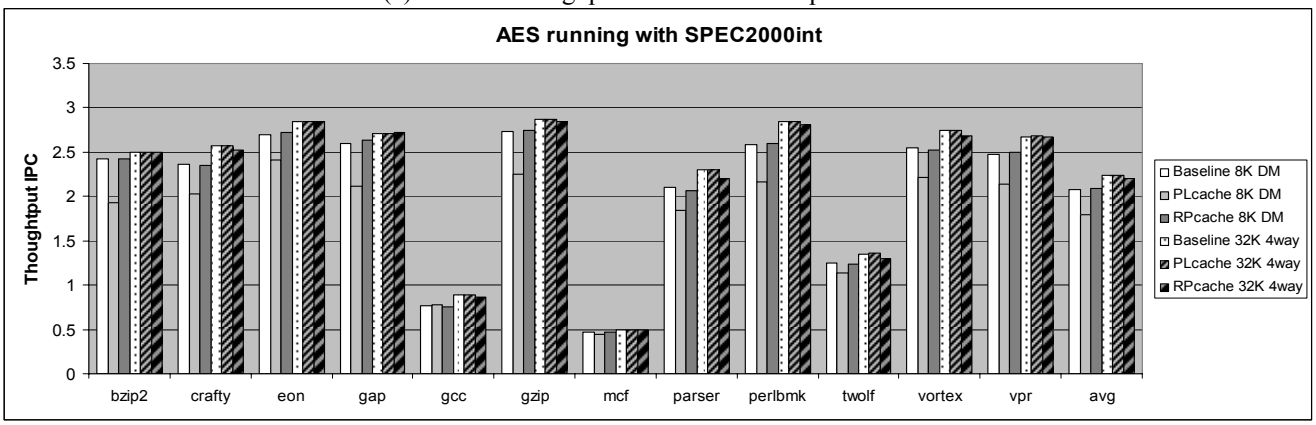


Figure 9. Performance comparison of AES code



(a) Overall throughput with SPEC2000fp benchmarks



(b) Overall throughput with SPEC2000int benchmarks

Figure 10. Performance impact on overall throughput

simulated program performs the generation of 1 KByte packets and the encryption of the packets, and runs alone on the processor. To examine the effects of the cache capacity and the configuration on performance, we vary the cache size from 4K to 32K and simulated the direct-mapped, 2-way and 4-way set-associative configurations for each size. Our results show that PLcache is sensitive to the cache size and configuration. When the size of the protected memory (5KB) is larger than the cache capacity (4KB cache), the performance is always bad because all cache lines are locked. Implementing the PLcache as a direct-mapped cache is also not a good idea since once a line is locked, it generates a lot of conflict misses. For cache sizes larger than the protected data, with set-associativity at least 2, the PLcache can achieve comparable performance to the traditional cache. In contrast, the RPcache consistently achieves almost the same performance as the traditional cache, regardless of the cache capacity and configuration. The performance impact caused by the random cache evictions in RPcache is negligible: worst case 1.7% (on 4K directed-mapped cache) and 0.3% on average. We also simulate the L2 PLcache and L2 RPcache. As the L2 cache is large enough to hold the working set, no performance degradation is observed.

Table 4. Simulation parameters

Simulation Parameters	Value
Decode/Issue width	4/4
Integer ALUs	4+1 multi/div unit
Floating-point ALUs	4+1 multi/div unit
ROB size	96
Physical RF size	96 each for Int/FP
Fetch Policy for SMT	icount
L1 instruction cache	64KB 2-way 32B
L2 unified cache	512K 8-way 64B
Cache access time	2 cycles L1, 12 cycles L2
Memory access latency	200 first chunk, 4 inter
L1 data cache ports	2
LSQ entries	48

Performance impact on the whole system due to the protected code: The PLcache and RPcache may impact the performance of the system during the execution of the protected code, e.g., the performance of other general purpose workloads running concurrently while encryption is being done for a file. In the simulation, we assume that the protected code (AES) is running with another thread simultaneously. We use an 8Kbyte direct-mapped L1 D-cache and a 32Kbyte 4-way L1 D-cache to bound the cache impact. The 6 bars per SPEC2000fp or SPEC2000int benchmark in Figure 10 show the simulations of the baseline, PLcache and RPcache for 8K 1-way L1 D-cache, then for 32K 4-way D-cache. For an 8Kbyte direct-mapped cache, PLcache causes an average performance degradation of 12% and 14% on floating point benchmarks and integer benchmarks, respectively. The RPcache causes 0.3% degradation on floating point benchmarks and 0.07% *improvement* on integer benchmarks. The improvement is a result of the swap operations of the RPcache which avoid many conflict misses. On a 32Kbyte 4-way cache, the PLcache achieves a 0.2% performance *improvement* on both integer and floating-point benchmark sets. This is because the 32Kbyte cache is large enough to hold the working sets for both threads and the protected code benefits from the locked cache lines that avoid misses on these lines. The performance degradation for the RPcache is 0.3% on FP suite and 1.2% on INT suite, respectively. The increase in performance degradation is due to the higher overhead associated with the swap operations for a set-associative cache. However, the absolute degradation is still very small. We also examined the effect of implementing the L2 cache as a PLcache or RPcache. The effect is again insignificant.

Although we only use AES as the protected code in our simulations, our conclusions are not specific to AES. The sensitivity of PLcache's performance to the cache configuration and capacity (relative to the size of the protected memory region) is due to the locking behavior and is not a result of any AES-specific factor. The robustness of the RPcache's performance is due to the fact that we allow sharing – and our design intentionally minimizes the restrictions on sharing.

5.3. Comparison with Prior-Art

Table 5 summarizes the advantages of our PLcache and RPcache solutions compared with the prior-art partitioned cache solution, in terms of both security and performance.

Table 5. Comparing with prior-art Partitioned Cache

Security & Performance	Partitioned Cache	Our PLcache	Our RPcache
Prevents external Interference?	Yes	Yes	Yes
Prevents Internal Interference?	No	Yes	Yes
Relative performance	Low	Medium	High

Security: All three approaches can prevent information leakage via external cache interference. Partitioned cache and PLcache provide private partitions to a process which are not accessible by other processes. RPcache randomizes the interference so that it carries no useful information. The partitioned cache can not, however, defend against attacks based on internal interference; a private partition still allows code within a process to contend for cache lines and cause interference, as in Bernstein's statistical attack. PLcache does not have this problem, because it explicitly locks the desired lines in cache, and other parts of the same process cannot interfere with these cache lines. RPcache randomizes the interference – hence it carries no useful information.

Performance: A partitioned cache does not allow a process which uses very few cache lines to make its unused cache lines available to other processes which may need more cache lines than they have in their partitions. Hence, it has the lowest performance among the three approaches. PLcache can achieve better performance because it has a locking mechanism that allows it to minimize the size of flexible private partitions, leading to better cache utilization. RPcache allows different processes to share cache slots and therefore has the smallest performance degradation. In addition, the performance of the partitioned cache and PLcache depend on software to specify proper partitioning of the cache, while the performance of the RPcache is very robust, with little dependence on the software and the underlying hardware cache architecture.

6. PAST WORK

The problem of information leakage via the cache was first mentioned and discussed in the context of covert channels [13] where the information is intentionally modulated over the cache interference. In 2002, Page [14] described a theoretical attack exploiting cache misses. In 2002 and 2003, Tsunoo et al. studied attacks against DES on computers with caches [15][16]. In 2005, Bernstein [5] and Osvik et al. [7] concurrently developed cache timing attacks against AES. Pervical [6] demonstrated an attack against RSA on an SMT processor. Since then, a number of new cache-based side channel attacks have been reported in [8][17].

Most past work on mitigating cache-based side channel attacks focused on software solutions. New implementations of AES and RSA were proposed and their performance evaluated [9]. Partitioning resources, including caches have also traditionally been used to mitigate covert channels; more recently, Page [18] also used a Partitioned Cache for mitigating side channels. Our PLcache uses a different approach to realize a minimal "virtual partition", achieving greater security *and* higher performance with little hardware cost (Table 5). We also propose a randomization-based cache solution which is completely different.

Other related work include the HIDE cache [19] which takes a probabilistic approach to mitigate control flow information leakage. In contrast, we focus on information leakage caused by cache interference rather than due to the exposure of address traces on the system memory bus. Our assumptions are fundamentally different and the proposed architectures are also very different.

7. CONCLUSIONS

Cache-based side channel attacks can be very dangerous. Almost all computing systems have shared caches at some level and will be susceptible to these attacks. These attacks are software attacks – very easy to perform, without the need for special equipment, and the attacker does not need physical access to the device. The attacker process can be unprivileged and can even be a remote client. The attacks are very effective: the full key bits can be recovered in a short time.

We presented analysis of why and how different types of cache-based side channel attacks work. We identify cache interference as the root cause of these attacks. We proposed novel general-purpose hardware solutions, the PLcache and the RPcache, that eliminate or randomize cache interference, respectively. The PLcache, with minimal hardware cost, can help the software developer achieve security without losing performance. With a little more hardware, the RPcache can robustly provide both security and performance, even without input from the programmer. Our performance evaluation shows that the RPcache causes performance degradation of less than 2% on average. Using an information-theoretic method, we mathematically proved the security of the RPcache.

Future work includes identifying more processor and cache induced side channel attacks, and finding solutions to mitigate this growing threat. We hope that this paper will help stimulate new research in the design of security-aware cache and computer architectures that do not sacrifice on performance, cost and energy consumption.

8. REFERENCES

[1] E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems", *Journal of Cryptology*, vol. 4, no. 1, pp.3-72, 1991.

[2] M. Matsui, "Linear Cryptanalysis Method for DES Cipher", *Advances in Cryptology – EUROCRYPT'93 (Lecture Notes in Computer Science no. 765)*, Springer-Verlag, pp. 386-397, 1994.

[3] Paul Kocher, Ruby B. Lee, Gary McGraw, Anand Raghunathan, and Srivaths Ravi, Security as a New Dimension in Embedded System Design, *Proceedings of the Design Automation Conference (DAC)*, pp. 753-760, June 2004.

[4] C. Kocher, J. Jaffe, and B. Jun. Differential power analysis, *Advances in Cryptology − CRYPTO'99*, vol. 1666 of Lecture Notes in Computer Science, pp. 388–397,1999.

[5] D.J. Bernstein, "Cache-timing Attacks on AES," available at: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[6] C. Percival, "Cache Missing for Fun and Profit," available at: http://www.daemonology.net/papers/htt.pdf

[7] D. A. Osvik, A. Shamir and E. Tromer, "Cache attacks and Countermeasures: the Case of AES", *Cryptology ePrint Archive*, Report 2005/271, 2005.

[8] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *SAC'06*, to appear.

[9] Ernie Brickell and Gary Graunke and Michael Neve and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR ePrint Archive*, Report 2006/052, Feb 2006.

[10] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. *Technical Report*, COMPAQ Western Research Lab, 2001.

[11] T. Cover and J. Thomas, "Elements of Information Theory," *John Wiley & Sons Inc.*, New York, 1991.

[12] M-Sim v2.0, http://www.cs.binghamton.edu/~jsharke/m-sim/

[13] Wei-Ming Hu, "Lattice scheduling and covert channels," *IEEE Symposium on Security and Privacy*, pp.52-61, 1992.

[14] Daniel Page, Theoretical use of cache memory as a cryptanalytic side-channel, *Technical Report CSTR-02-003*, Department of Computer Science, University of Bristol, 2002.

[15] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, Hiroshi Miyauchi, "Cryptanalysis of block ciphers implemented on computers with cache," *Proc. International Symposium on Information Theory and its Applications*, pp.803-806, 2002.

[16] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, Hiroshi Miyauchi, "Cryptanalysis of DES implemented on computers with cache," *Proc. CHES 2003*, LNCS 2779, 62-76, 2003.

[17] Onur Aciçmez, Werner Schindler, and Çetin Kaya Koç, Cache Based Remote Timing Attack on the AES, to appear in *RSA Conference 2007*, Cryptographers' Track.

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

[19] X. Zhuang, T. Zhang, and S. Pande, "HIDE: an infrastructure for efficiently protecting information leakage on the address bus," *ACM 11thInternational Conference on Architecture Support for Programming Language and Operating Systems*, 2004.