Secure Cache Modeling for Measuring Side-channel Leakage

TIANWEI ZHANG AND RUBY B. LEE
Princeton University

Abstract

Side-channel attacks try to break a system's confidentiality using physical information emitted from the targeted system. Such information is leaked out through cache side channels, which can exist in many parts of the system. Cache memories are a potential source of information leakage through side-channel attacks, many of which have been proposed. Meanwhile, different cache architectures have also been proposed to defend against these attacks. Thus it is necessary to evaluate the effectiveness of the proposed defense approaches.

In this paper, we propose two methods to evaluate a system's vulnerability to cache side-channel attacks. First, we run actual attack programs and recover the cipher keys to directly show if the target system is attackable through such side-channel attacks. We also provide a new key-vote metric to quantify the system's vulnerability to the attack. The actual attack is accurate, but is slow and cipher specific. Hence, we propose a second method based on new models of cache architectures and their information leakage potential. We define a novel Interference Matrix to evaluate a system's vulnerability to entire categories of cache side-channel attacks, rather than to a specific attack. These models can give more comprehensive conclusions on a system's vulnerability to side channel attacks. Finally we check whether the two methods give consistent results.

1 Introduction

Confidentiality is a major concern in information security. One solution for the confidentiality problem is to use strong cryptography. With ciphers, the encrypted data or secrets can be freely sent over public networks or stored in publicly-accessible storage, as long as the encryption key is kept secret and accessible only by authorized parties. In modern cryptography, Kerckhoff's principle states that a cryptosystem should be secure even if everything about the system is public knowledge except the key. So the confidentiality of the encryption key is particularly important: if the encryption key is leaked, then the confidentiality protection provided by strong encryption is nullified.

Ingenious cryptographic algorithms have been designed to enhance the strength of ciphers. However, different attacks have been proposed to break these ciphers and recover the keys. The simplest one is a brute-force attack, which tries all the possible keys. It requires exponential time and energy, and is computationally infeasible if the key is long enough. Other mathematical attacks include differential cryptanalysis [1] or linear cryptanalysis [2]. These attacks target the weakness of the cryptography algorithms and try to recover the keys through mathematical analysis, which is usually specific to each cipher.

Unlike the attacks mentioned above, side-channel attacks are information leaks through a medium which is not intended for communications. This medium is called a side-channel. Side channels convey physical characteristics of the system and leak secret information indirectly.

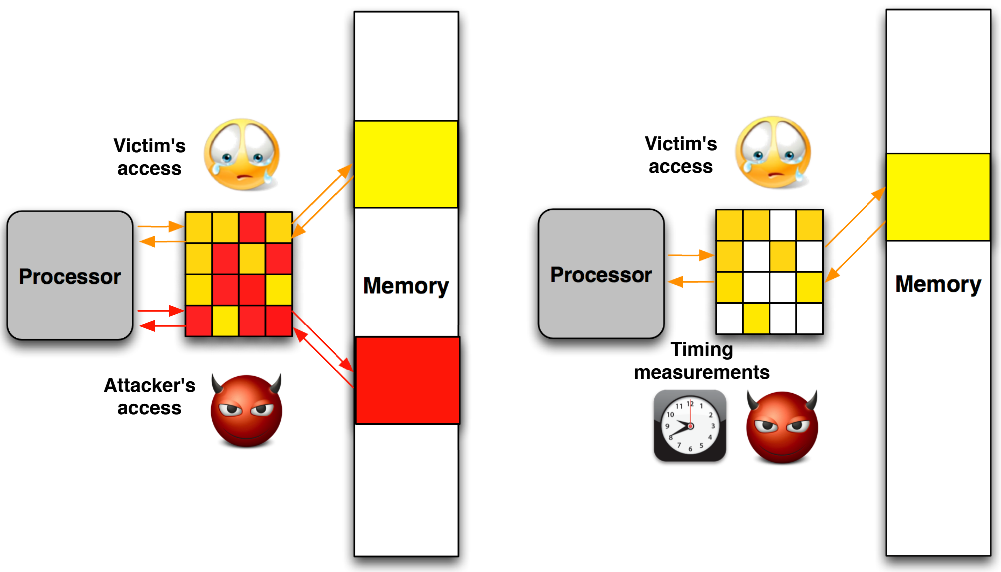
Hardware-based side-channel attacks are very hard to defend against for several reasons. First, side-channel attacks target the vulnerabilities of the system instead of the cryptographic algorithms. The same attack strategy can often be applied to different ciphers. Second, side-channel attacks can be successfully performed in a short period of time (e.g., average 3 minutes in [3]), which will not cause any noticeable impact on the system. Third, the attackers do not need high privileges to launch an attack. All the operations are within their authorized privilege level. They only need to know the plaintext (known or chosen plaintext attack), or just the ciphertext, or even neither of them [3, 4]. Fourth, side-channels exist widely in different systems. Power [5, 6], electromagnetic radiation [7, 8], timing [9], etc., all can be observed by the attacker to infer the inaccessible critical information. Fifth, the observable side-channel information is due to the natural physical features of the system, so it is very difficult to eliminate these side-channels. Particularly worrisome is the fact that performance and power optimization features in modern processors are a source of side-channel leaks. For example, the use of the caches in the memory system can significantly reduce the effective memory access time, and is one of the most important features for improving performance in a modern processor. However, the different access time characteristics due to cache hits and misses also provide attackers a source of side-channel information leakage. It is typically unacceptable to disable the cache (to disable such side-channels), because of the severe performance degradation that would result.

Since side-channel attacks exploit the physical features of systems, many defense strategies focus on security improvements of the system implementations. For instance, to defend against cache side-channel attacks, a variety of secure cache architectures have been proposed in recently years [10, 11, 12]. These cache architectures aim to thwart certain types of side-channel attacks without huge performance cost. The performance of these architectures can be tested by different benchmarks, but their security effectiveness have only been analyzed qualitatively. Thus, general quantitative methods of measuring the potential side-channel information leakage are desirable when trying to compare different cache architectures. Such methods are important because they can reveal which features of the system are more prone to leaking critical information, and contribute to the trade-off analysis between performance, power and security of different cache architectures.

In this paper, we propose two quantitative evaluation methods for a system's vulnerability to cache side-channel attacks. For clarity, we focus on one class of attacks: the one induced by cache misses due to external interference (described in section 2). With certain modifications, these methods can be applied to other types of cache side-channel attacks. The main contributions of this paper are:

- Proposal of a first evaluation method using an actual attack program with a new key-byte recovery metric;
- Proposal of a second evaluation method based on new state machine modeling of cache architectures and computation of an Interference Matrix.

Section 2 gives the background of cache side-channel attacks and various cache defenses. Section 3 describes the first evaluation method and its results. In section 4, we build abstract models of cache architectures and define the Interference Matrix to measure the information leakage. Section 5 compares the two methods, their scope and the consistency of their results. Section 6 discusses related work. Section 7 gives our conclusions and suggestions for future work.



(a) Access-based attack (b) Timing-based attack

Figure 1: Concept of cache side-channel attacks

2 Background

2.1 Cache Side-channel Attacks

Side-channel attacks try to break a cipher through by-product information from side-channels. Among all the potential sources of information leakage, cache side-channels are particularly dangerous, as caches exist in essentially all modern processors, from embedded systems to cloud servers. Besides, many cipher programs involve memory accesses that depend on the encryption keys, so the usage of the memory (thus the cache) gives the attacker the chance to break the ciphers through cache side-channel attacks.

As programs execute on the system, they may have different cache behaviors (hits or misses) when accessing the memory. These behaviors have different timing characteristics. The attackers try to capture these timing characteristics, and then deduce the victims' memory accesses that might help them finally break the ciphers. They have different ways to realize this goal: if the attacker shares the caches with the victim, he can measure his own cache access time to test the state of the cache, thus to infer the victim's cache accesses. This is called an access-based attack [13, 3], shown in figure 1a. The attacker can also measure the victim's execution time to deduce the victim's cache hits or misses during his execution, to infer his memory accesses. This is called a timing-based attack [14, 15], shown in figure 1b.

2.2 Attack Categories

A large number of side-channel attacks based on caches have been proposed during the past few years [13, 14, 16, 15, 4, 17, 3]. The root cause of all such attacks is due to interference: either external interference between the attacker's program and the victim's program, or internal interference inside the victim's own program [10]. Combined with the cache behaviors the attackers want to observe (cache misses or hits), we have the four cache side-channel attack categories [18] shown in Table 1 and described below.

Type I: Cache Misses due to External Interference. In this class, the attacker and the victim

Table 1: Cache side-channel attack categories

	External Interference	**Internal Interference**
Cache Misses	I. Access-based attacks e.g., Percival's attack	III. Timing-based attacks e.g., Bernstein's attack
Cache Hits	II. Access-based attacks e.g., Shared library	IV. Timing-based attacks e.g., Bonneau's attack

run their processes on the same processor, and they share the same data cache. So the victim's process may evict the cache lines holding the attacker's data, which will cause the attacker future cache misses and give the attacker the chance to infer the victim's cache accesses. Some access-based attacks belong to this class, and a typical one is Percival's attack [13].

The attacker usually adopts the technique of "Prime and Probe" [4] to perform this kind of access-based attacks. In this method, the attacker assigns his program a contiguous byte array, the size of which is equal to the cache size. In the "Prime" stage, he reads every memory block in the array. Then the cache is fully occupied by the attacker. After a certain time interval, the attacker performs the second "Probe" stage: he again reads each block in the array, and measures the access time of each cache set. A large access time means the attacker has a cache miss, indicating that this cache set has been accessed by the victim during that interval, and the attacker's data in that set has been evicted out of the cache by the victim. By inferring the state of the cache for the victim's execution between the Prime and Probe stages, the attacker can get to infer the cache accesses of the victim's program. Then he can analyze such side-channel information to recover critical confidential data, such as an encryption key used by the victim.

Type II: Cache Hits due to External Interference. In this class, the attacker and the victim share some memory space (e.g, a shared cryptography library). First, the attacker evicts all the shared memory blocks out of the cache. After a certain time interval of the victim's execution, the attacker reads every shared memory block and measures the access time. A small time means the attacker has a cache hit, indicating that this cache line has been accessed by the victim during that interval and re-fetched into the cache by the victim. Then the attacker can infer the memory addresses the victim has accessed. The access-based attack in [3] belongs to this class.

Type III: Cache Misses due to Internal Interference. In this class, the attacker does not run programs simultaneously with the victim. Instead, he only measures the total execution time of the victim, e.g., for encryption of one plaintext block. A longer execution time indicates there may be more cache misses from the victim's own execution (which includes its wrapper code); this can give the attacker some information about the victim's memory accesses. Some timing-based attacks belong to this class, such as Bernstein's attack [14].

Type IV: Cache Hits due to Internal Interference. Similar to the above attack, in this class, the attacker still only needs to measure the total execution time of the victim. But he only cares about cache hits inside the victim's code. If the attacker measures a shorter execution time, it may be due to more cache hits during the victim's execution. So the attacker may be able to infer information about the encryption keys through "cache collision" (i.e., cache hits) of memory accesses. Some timing-based attacks belong to this class, such as Bonneau's attack [15].

In this paper, we focus on category I cache attacks, which are induced by cache misses due to external interference. These attacks require less time to succeed than the internal interference attacks (categories III and IV), and they do not need to share some memory addresses with the victim (as in category II attacks). Hence, they are often considered the most powerful cache side-channel attacks.

2.3 Cache Defenses and Architectures

In this section, we discuss proposed cache defenses and secure cache architectures. We describe two general approaches to protect against cache side-channel attacks: *Isolation* and *Randomization*. Then we give some examples of secure caches that adopt these two approaches.

2.3.1 Isolation.

The reason that caches can be exploited as side-channels in the external interference attacks is the attacker and the victim can share the caches. So one straightforward approach to prevent information leakage is to prevent the cache sharing between the attacker and the victim, by dividing the cache into different zones for different processes. We have the following cache designs using this idea:

Static-Partitioning (SP) cache: This cache is statically divided into two parts either by ways or by sets. In set-associative caches partitioned by ways, each way is reserved for either the victim or the attacker program. The cache can also be partitioned by sets, where each set is reserved for either the victim or the attacker program. Due to the elimination of cache line sharing, SP caches can effectively prevent external interference, but at the cost of degrading the performance because of the static cache partitions. (Note that allowing the partitioning to be adjusted dynamically, rather than only statically, can leak some information.)

Partition-Locked (PL) cache: PL cache [10] uses a finer-grained dynamic cache partitioning policy. In PL cache, each memory address has a protection bit to represent if this memory block needs to be locked in the cache. Once the protected block (e.g., the victim's critical data) is locked in the cache, it can not be replaced by an unprotected block (e.g., the attacker's data). Instead, the attacker's data will be directly sent between the processor and the memory, without filling the cache. This replacement policy will thwart the attacker's plot to spy on the victim's cache accesses. If all the sensitive data are pre-loaded before encryption starts, then PL cache enables constant-time memory accesses, since all accesses to sensitive data will result in cache hits.

2.3.2 Randomization.

In this approach, side-channel information is randomized, thus no accurate information is leaked out from caches. There are at least two ways to realize randomization: adding random noise in the attacker's observations and randomizing the mappings from memory addresses to cache sets.

Random-Eviction (RE) cache: a RE cache periodically selects a random cache line to evict. This can add random noise into the attacker's observations so he cannot tell if an observed cache miss is due to the cache line replacement or the system's random eviction. This will increase the attacker's difficulty in recovering secret information like a cipher key.

Random-Permutation (RP) cache: RP cache [10] uses random memory-to-cache mappings to defend against cache side-channel attacks. There is a *permutation table* (PT) for each process. This enables a dynamic mapping from memory addresses to hardware-remapped cache sets. When one process **A** wants to insert a new block D into the cache, it checks **A**'s *permutation table* and finds a victim block R in set S. If this R belongs to another process **B**, instead of evicting R, thus revealing information to outsiders, a random block R' in a random set S' is selected, evicted and replaced by D. At the same time, the sets S and S' in **A**'s *permutation table* are swapped, and the blocks in these two sets belonging to **A** are invalidated. Since the victim's memory-to-cache mappings are dynamic and unknown to the attacker, the attacker cannot tell which lines the victim actually accessed. This is different from conventional caches, which have static and fixed memory-to-cache mappings, rather than dynamic and randomized mappings.

NewCache: NewCache [11, 19] also randomizes the memory-to-cache mappings, but does this by introducing a *Logical Direct-Mapped Cache (LDM)*, which does not physically exist. The mapping from memory addresses to the LDM cache is direct-mapped, with the benefits of simplicity and speed or power efficiency. The mapping from the LDM cache to the physical cache is fully-associative and random. This dynamic and random mapping can enhance the security against information leakage. For the replacement policy, if the incoming block D cannot find any block in the physical cache with the same index (called an index miss), it will randomly choose a block R to replace. If the incoming block can find a block R in the physical cache with the same index, but the tag is different (called a tag miss), then D may replace R (when either D and R are both protected or both unprotected) [19].

The questions we want to answer in this paper are: "Do these secure caches really defend against cache side-channel attacks?" and "What are the relative vulnerabilities of different cache architectures to side-channel attacks?". In the next two sections, we will use two different methods to evaluate the effectiveness of these cache designs for preventing attacks induced by cache misses and external interference.

3 Actual Attack Programs and Key Recovery

In this section, we describe the first evaluation method –running the actual attack program. This method is straightforward as the success or failure of the attack can directly reflect the system's vulnerability to this attack. We launch a synchronous "Prime and Probe" attack on the AES cipher [4]. Then we try to recover the cipher keys, and use the votes of candidate keys to quantitatively show the feasibility of this attack, and hence the system's vulnerability.

3.1 AES -the Advanced Encryption Standard

AES is a powerful symmetric-key cipher, which is widely used throughout the world. It is an iterated block cipher with block sizes of 128, 192 or 256 bits. For encryption, the plaintext block is converted into the ciphertext block with several rounds. Each round consists of four functions: SubBytes, ShiftRows, MixColumn and AddRoundKeys. These functions include linear and nonlinear operations, e.g., shifting or substitution, which consume computation resources.

To efficiently implement AES, most systems adopt the table lookup technique. Several tables are precomputed and stored in memory before the encryption. At each round, instead of calculating the four functions, the pre-computed tables are looked up to get the output of this round. This can have an order of magnitude performance improvement over the separate algebraic computations. However, accessing the memory (thus accessing the cache) provides the attacker a potential side-channel to observe which entries in the AES lookup tables are used during the encryption. Thus, the use of lookup tables in memory is the cause of many cache side-channel attacks on software AES implementations.

3.2 Attack Strategy

We consider AES encryption with an encryption block size of 16 bytes. Assume the plaintext is $\mathbf{P} = \{p_0, \ldots, p_{15}\}$ and the encryption key is $\mathbf{K} = \{k_0, \ldots, k_{15}\}$. The attacker can use the synchronous "Prime and Probe" technique [4] to recover the key $\mathbf{K}$. Before encryption, the attacker executes the "Prime" stage to fill the cache with his own data. Then it is the victim's turn to encrypt one block of plaintext **P**. The victim calculates $x_i = p_i \oplus k_i$ $(i = 0, \ldots, 15)$ as the index to access the lookup tables in the first round. If the table entry indexed by x_i is mapped to the cache set s_i, the victim will occupy one cache line in set s_i. After the encryption, the attacker will take over to execute the "Probe" stage. He will observe a cache miss in accessing set s_i.

The attacker does not know the mappings from x_i to his observation of cache miss or hit for s_i, but he can perform two phases to recover the key: the *study* phase and the *attack* phase. In the *study* phase, the attacker provides a known key **K** and plaintext **P** to the system for encryption. So the index $x_i = p_i \oplus k_i$ is mapped to cache set s_i. In the *attack* phase, the system will encrypt the plaintext **P′** provided by the attacker with key **K′**, which the attacker does not know and wants to recover. So the index $x'_i = p'_i \oplus k'_i$ is mapped to cache set s'_i. If in the two phases the victim accesses the same table entry ($x_i = x'_i$), the attacker will observe cache misses in the same set ($s_i = s'_i$), which indicates $p_i \oplus k_i = p'_i \oplus k'_i$. Then the attacker can easily recover the unknown key with the equation below:

$$k'_i = p'_i \oplus p_i \oplus k_i \tag{1}$$

The attacker provides a large quantity of plaintexts to the victim for encryption. For each plaintext, the attacker performs the "Prime and Probe" attack, and measures the access time of each set. After all the encryptions are done, the attacker calculates the average access time for each cache set s, for each of the 16 plaintext bytes i, for each byte value b. Algorithm 1 shows the pseudo code for generating such access time profiles. (We have 16 plaintext bytes, each byte has 256 possible vaues (0, 1, ...,$2^8 - 1$), and we assume the cache has 128 sets.)

Algorithm 1 Generating profiles for each phase

```
sum[16][256][128]: sum of access time for each byte, value and set
count[16][256][128]: sum of counts for each byte, value and set
total_sum[128]: sum of access time for each set
total_count[128]: sum of counts for each set
avg[16][256][128]: normalized average access time for each byte, value and set

for (each plaintext p) do
    for (each set s of 256 cache sets) do
        let t = time measurement of set i
        for (each byte i of 16 plaintext bytes) do
            let b=value of byte i of p
            total_sum[s] = total_sum[s]+t
            total_count[s] = total_count[s]+1
            sum[i][b][s] = sum[i][b][s]+t
            count[i][b][s] = count[i][b][s]+1
        end for
    end for
end for
for (each set s) do
    let t_avg = total_sum[s] / total_count[s]
    for (each byte i) do
        for (each value b) do
            avg[i][b][s] = sum[i][b][s] / count[i][b][s] - t_avg
        end for
    end for
end for
return avg[16][256][128]
```

3.3 Implementation

We use gem5 [20, 21] to implement cache systems and launch the above side-channel attack. The system configuration we use for all the experiments in section 3 is summarized in Table 2. We simulate L1 caches, with cache size of 32KB, line size of 32B and set-associativity of 8-way, which are standard configurations in modern processors. The victim runs AES encryption for 2^{18} random blocks of 16 bytes, in two phases. In the *study* phase, the known key is all zeros, i.e., $k_i = 0x00, (i = 0, \ldots, 15)$. In the *attack* phase, the key is unknown to the attacker. The attacker primes the cache before encrypting one block and probes the cache after this block encryption, collecting the access time of each set. Then based on Algorithm 1 he can generate the three-dimensional average access time array, **avg**[16][256][128] for each phase.

Table 2: Configurations in gem5 implementation.

Parameter	Value
CPU-type	out-of-order
ISA	x86
Execution mode	system call emulation
CPU clock	2GHz
# of cache levels	1
Cache size	32KB
Line size	32B
Associativity	8-way

3.4 Attack Results

The results from the attacker's measurements can visually show if the attack is successful or not, even without needing the off-line phase to recover the keys. Since each of the 16 bytes of plaintext and key are independent, we can consider one byte at a time. We draw a two-dimensional grayscale figure to show the average access time. Figure 2 shows the **avg**[14][b][s] of a conventional cache in the *study* (left) and *attack* (right) phases for byte 14 (the other bytes have similar results). In the two figures, the vertical axis is value b (0-255) of byte 14 and the horizontal axis is the cache set s (0-127). The lighter pixels in (s, b) denote a larger average access time, meaning the attacker observes a cache miss of set s, when the value of plaintext byte 14 is b. Thus the attacker can infer that the victim may have accessed a table entry located in set s when encrypting this block.

From figure 2 we can clearly see some light line patterns in both phases. In the *study* phase, because the key is all 0, $x_{14} = p_{14} \oplus k_{14} = p_{14}$. In a conventional cache, x_{14} is approximately linearly mapped to the cache set s_{14}. Thus the cache set s_{14} is approximately linearly mapped to p_{14}. That is why we can see one diagonal line in the study phase. In the *attack* phase, as the key is non-zero, some segments of the line are shifted due to the operation of XOR between the key and the plaintext. The patterns in the two phases can reveal the unknown key-byte k_{14} to the attacker. So *this attack on the conventional cache succeeds*.

3.4.1 Isolation.

We consider the attack results for the isolation approach. Figures 3 and 4 show the access time profiles for SP cache and PL cache. From these figures, the attacker cannot detect lighter patterns, unlike in figure 2. The vertical dot lines in some sets of the two figures are due to the cache footprints, and cannot leak critical information. So it is impossible to recover the keys from these two phases. Thus *the attack on the SP cache and the PL cache fails*.

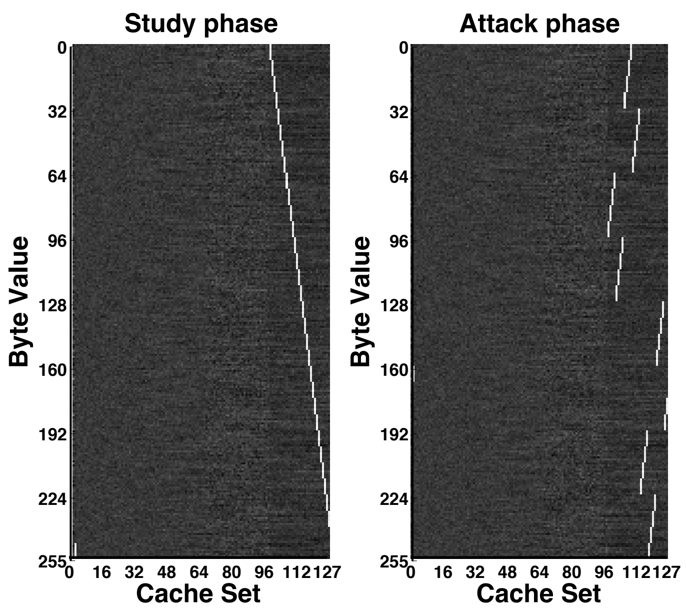


Figure 2: The result of "Prime and Probe" attack on a conventional cache in the *study* phase (left) and *attack* phase (right)

3.4.2 Randomization.

Next we consider the randomization approach. Figures 5 and 6 show the results of RE-1000 cache and RE-10 cache (randomly evict a line every 1000, or every 10, memory accesses, respectively). We can still see line patterns in the two caches (The pattern in figure 6 is not very clear since more noise is introduced to the attacker's observations as more frequent random evictions are performed). So *the attack on RE cache succeeds*.

Figures 7 and 8 show the results of RP cache and NewCache. The two figures do not show any light line patterns to the attacker. Thus no information about the encryption keys is leaked out. *The attack on RP cache and NewCache fails.*

3.5 Key Recovery Analysis

Detection of patterns can only qualitatively tell us if the attack is successful or not. In order to compare the vulnerability of different caches, we try to recover the keys based on the attack results, and then use the number of votes of the candidate keys to quantitatively reveal the system vulnerability to this attack.

Algorithm 2 describes the algorithm to recover keys. For each byte i of the plaintext in the *study* phase, we consider each value b of this byte and find the set s with the maximum access time ($\operatorname{argmax} avg[\mathbf{i}][\mathbf{b}][\mathbf{y}]$). This means the attacker observes the victim's access to set s when the value of plaintext byte i is b. Then in the *attack* phase, the attacker scans the 256 possible byte values and finds the ones b' which have the maximum access time in the same set ($s = s'$). So with the unknown key, the attacker observes the victim's access to the same set when the value of plaintext byte i is b'. It may be possible the two accesses refer to the same table entry. Then based on equation 1, the attacker can find a candidate key-byte.

After the attacker considers all the byte values in the *study* phase, the ideal case is that each candidate key-byte generated is the correct one. However, due to measurement noise, some false key-bytes may also be generated as candidates. To distinguish the correct key from the false ones, **we define the vote for key-byte as the number of generations of that key-byte value during the key recovery process.** Algorithm 2 gives the votes of each candidate key-byte

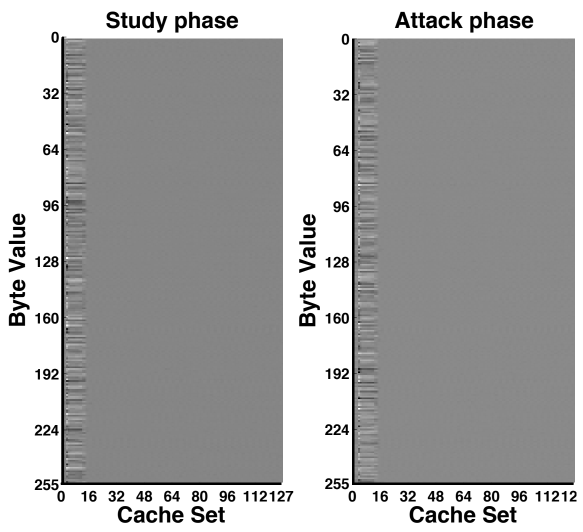


Figure 3: SP Cache

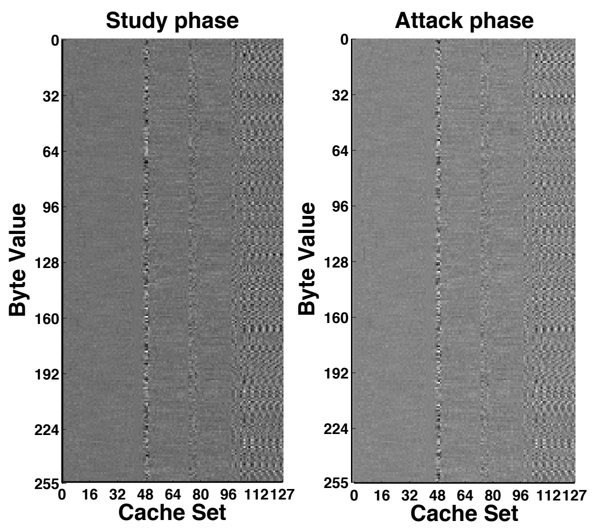


Figure 4: PL Cache

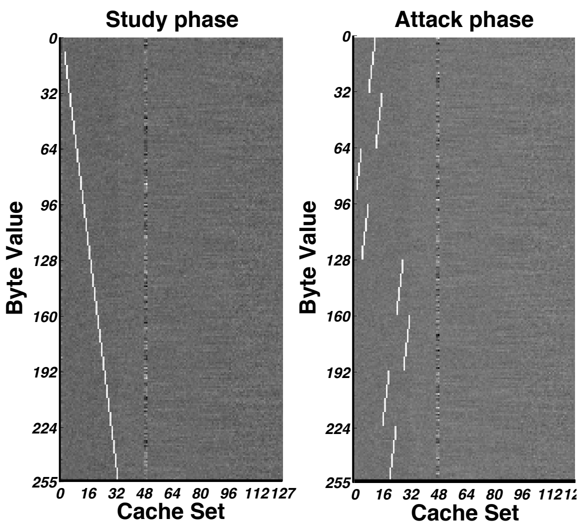


Figure 5: RE-1000 Cache

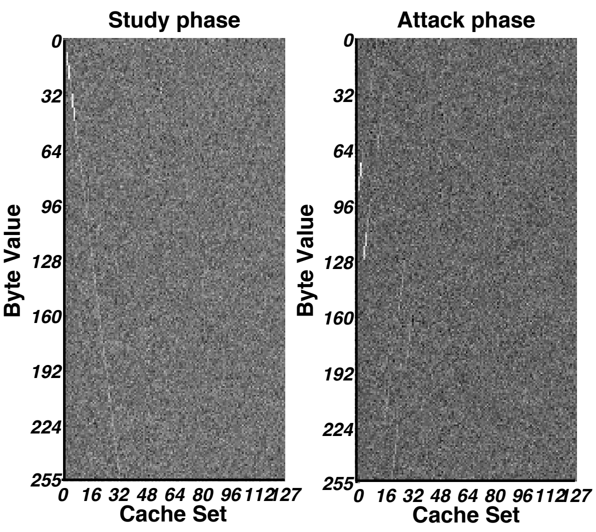


Figure 6: RE-10 Cache

for each byte value. In a successful attack, the correct key-byte value should have many more votes than the false ones. If the correct key cannot be distinguished from the other candidates, the attacker will be unable to recover it. We show the recovery result of key-byte 14 in the next few sections.

3.5.1 Cache parameters.

Let's first consider the effects of cache parameters on the feasibility of the attack. Figure 9 shows the votes of key-byte 14 candidates for conventional caches with different cache sizes (32KB/16KB), associativities (4-way/8-way), and line sizes (32B/64B). The horizontal axis is the key-byte candidate from 0 to 255, and the vertical axis is the votes for each candidate. Figure 9a shows the 32KB cache size, 8 way set-associative and 32B line size conventional cache. We can see that eight key candidates (32-39) get more than 230 votes while the rest are close to zero. The correct key-byte value 35 is among the top eight candidate keys, but the attacker cannot pick it out. This is clear because one cache line contains 8 AES entries. When the attacker observes

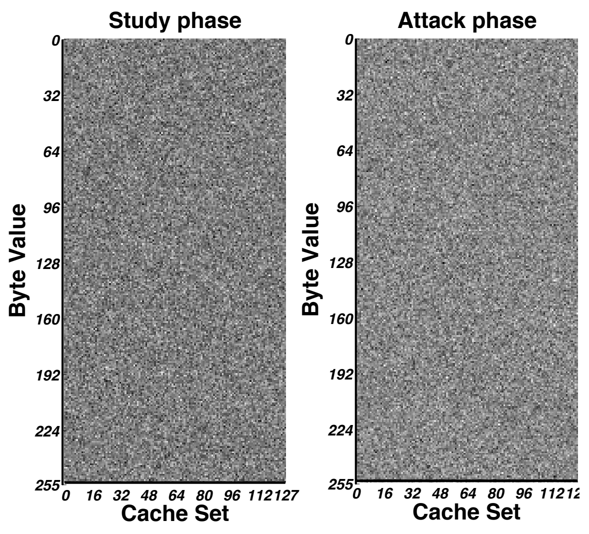


Figure 7: RP Cache

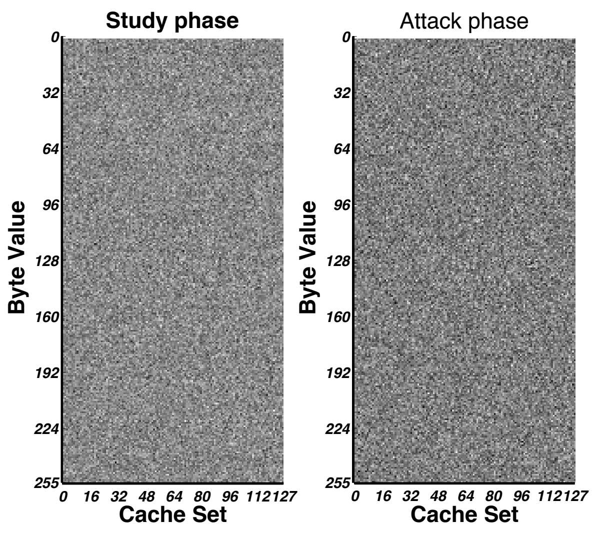


Figure 8: NewCache

Algorithm 2 Generating votes of each candidate key

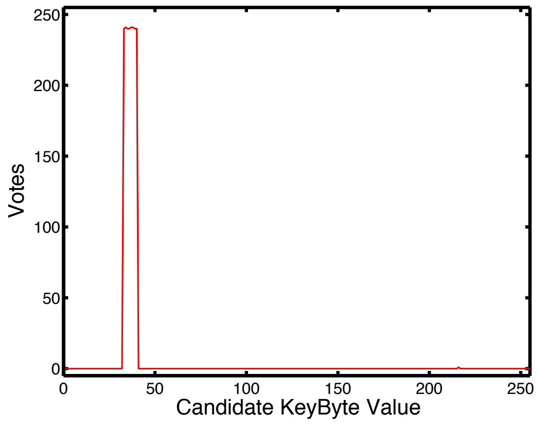
```
key_votes[16][256]: votes of key candidate for each key-byte value

for (each byte i of 16 plaintext bytes) do
    for (each value b of 256 byte values) do
        let s = $\underset{y}{\operatorname{argmax}}\, avg[\mathbf{i}][\mathbf{b}][\mathbf{y}]$
        for (each value b′ of 256 byte values) do
            let s′ = $\underset{y'}{\operatorname{argmax}}\, avg'[\mathbf{i}][\mathbf{b'}][\mathbf{y'}]$
            if ( s = s′ ) then
                $k'_i = b' \oplus b \oplus k_i$
                key_votes[i][$k'_i$] = key_votes[i][$k'_i$] + 1
            end if
        end for
    end for
end for
return key_votes[16][256]
```

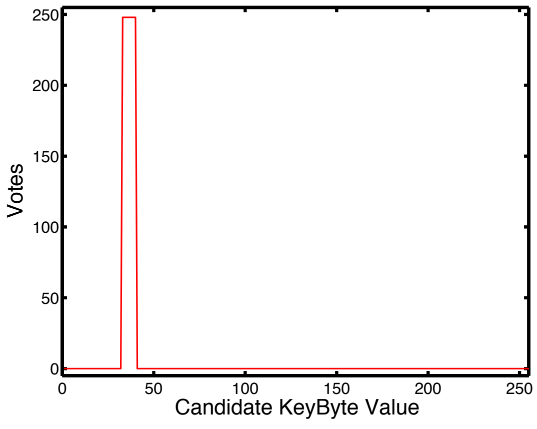
the victim's access to one cache line, he is unable to differentiate which AES entry is actually accessed. He needs other methods like brute-force to find the correct key from the 8 possible values -which is not hard to find. If we compare different associativities (figures 9a and 9b) and different cache sizes (figures 9a and 9d), there are no huge differences on the feasibility of the attack. If we compare different line size (figures 9a and 9c), we can see 64B line size cache has 16 top (32-47) candidate keys which the attacker cannot differentiate, as one cache line contains 16 AES entries. So it will be more difficult for the attacker to recover the correct key. For the vulnerability of the different cache parameters, we have the following relation: 32B line size $>$ 64B line size

3.5.2 Isolation.

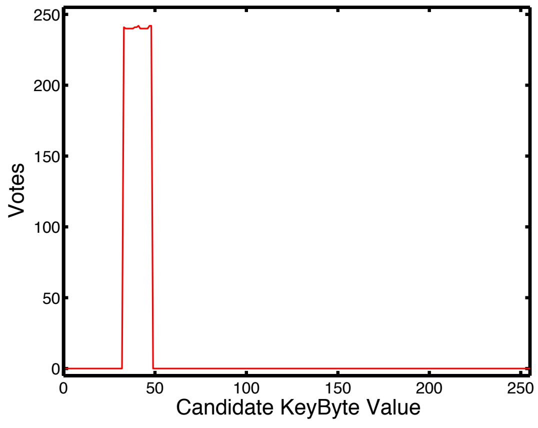
We study the key recovery results of SP cache and PL cache, displayed in figures 10 and 11. From these two figures, we see that neither the SP or PL caches have distinguished candidate keys revealed to the adversary. So we have the following relation for the vulnerability of isolation approaches: Conv $>$ $\{SP, PL\}$



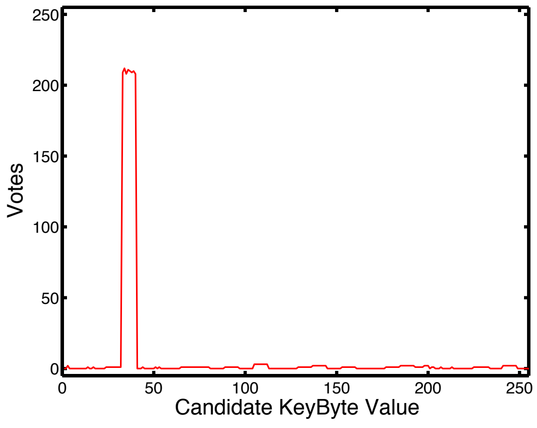
(a) 32KB size, 8 way, 32B line size



(b) 32KB size, 4 way, 32B line size



(c) 32KB size, 8 way, 64B line size



(d) 16KB size, 8 way, 32B line size

Figure 9: Votes of key-byte candidates for conventional cache.

3.5.3 Randomization.

The key recovery results of the randomization approaches, for RE-1000, RE-10, RP cache and NewCache, are shown in figures 12, 13,14 and 15. We see that RE-1000 cache can leak eight candidate keys to the attacker, similar to the conventional cache. RE-10 cache can also leak eight candidate keys, but they have fewer votes than a conventional cache and RE-1000 cache. This indicates that a RE cache with more frequent random evictions is more difficult to attack. For RP cache and NewCache, no candidate keys are leaked to the attacker, It is almost impossible for the attacks on these two caches to succeed, and they are very effective at defending against this type of side-channel attack. The relative vulnerabilities of these caches to this attack is: $\{\text{Conv}, \text{RE-1000}\} > \text{RE-10} > \{\text{RP}, \text{New}\}$

4 New Models of Cache Architectures and Interference

Using an actual attack program, as in section 3, can accurately reflect the system's vulnerability to side-channel attacks. However, it has some limitations. First, it has a narrow scope as it only applies to a specific cipher algorithm. Second, running an actual attack usually takes a long time (in the gem5 simulator, it takes 30 to 70 hours to tell if the attack is successful on one cache

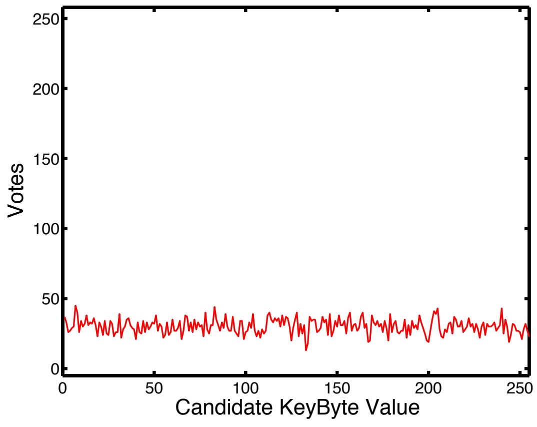


Figure 10: SP cache.

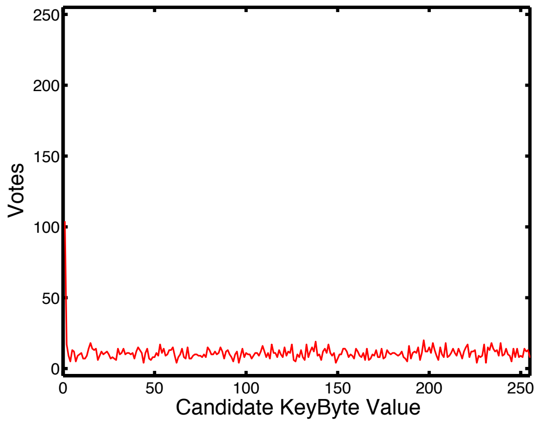


Figure 11: PL cache.

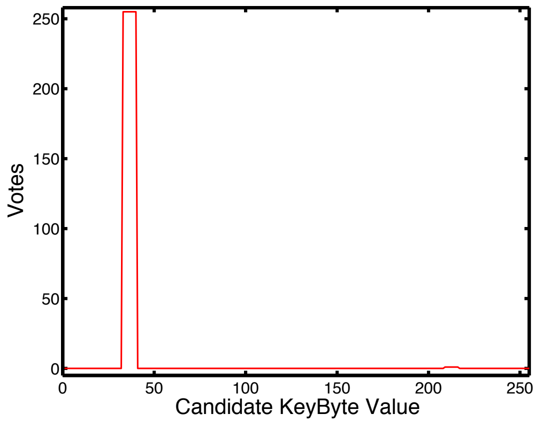


Figure 12: RE-1000 Cache

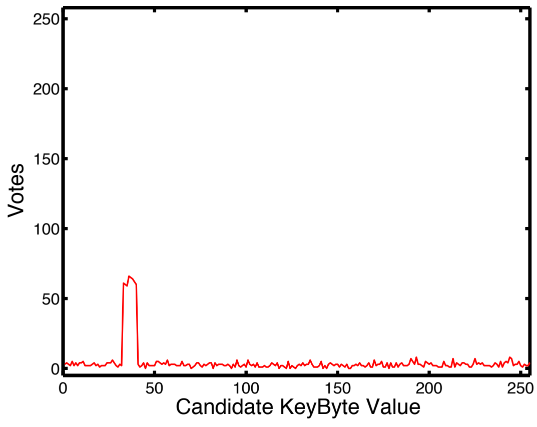


Figure 13: RE-10 Cache

configuration). Hence, we hope to find a faster and more general approach to measure the cache side-channel leakage of different cache architectures. In this section, we build abstract models of cache architectures which focus on the essential features of caches that can be exploited to leak information. For clarity, we focus on category I cache attacks (observing cache misses due to external interference) in Table 1, and the goal with our new cache modeling technique is to try to cover all attacks in category I on all ciphers, rather than only the specific "Prime and Probe" attack on AES described in section 3. Our modeling methodology can also be extended to apply to the other categories of cache attacks.

4.1 General Model of Side-channel Leakage

We first look at a model of information leakage in a generic system. We consider a system which can be modeled as a finite-state machine (FSM), with different states, and transitions between states under some predefined rules. The states in the FSM are modeled as either high or low confidentiality. We use Bell-LaPadula (BLP) [22] Multilevel Security (MLS) [23] policy to describe the information leakage in this system. To protect the system's confidentiality, the BLP security policy enforces two properties for the system: (1) a subject (e.g., program) at a low confidentiality level cannot read data classified at a higher confidentiality level ("No read up"), and (2) a subject at a higher confidentiality level cannot write data to a lower confidentiality level ("No write down"). So the basic idea in the BLP security policy is *"No information flows*

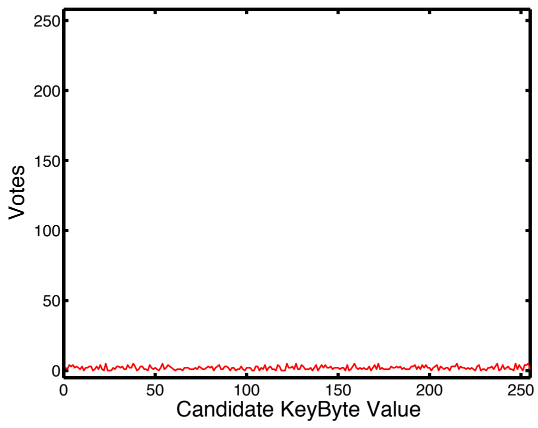


Figure 14: RP Cache

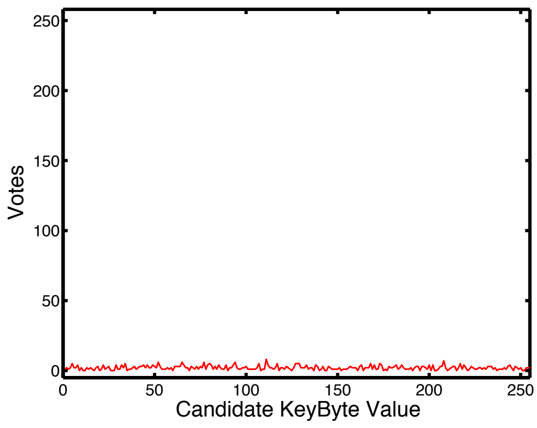


Figure 15: NewCache

from high confidentiality to low confidentiality". This can be used to model no information leaks from the victim (high confidentiality level) to the attacker (low confidentiality level).

Now, we show how to apply the BLP policy to model side-channel information leakage. The side-channel leakage is a statistical process. The attacker does the analysis combining all the information he collected during the attack process. So to evaluate the information leakage, we should consider the overall effects of information flow. Figure 16 shows two cases with no side-channel information leakage. In figure 16a, there is no information flow from the victim's high confidentiality level to the attacker's low confidentiality level. So there is no information leakage. In figure 16b, the attacker observes two streams of information flows. However, the two flows contain inconsistent or ambiguous information. The attacker cannot deduce accurate conclusions from these two flows. Hence, this can result in no useful side-channel leakage. So we postulate that **two criteria need to be satisfied in order for an attacker to acquire useful (or accurate) confidential information:**

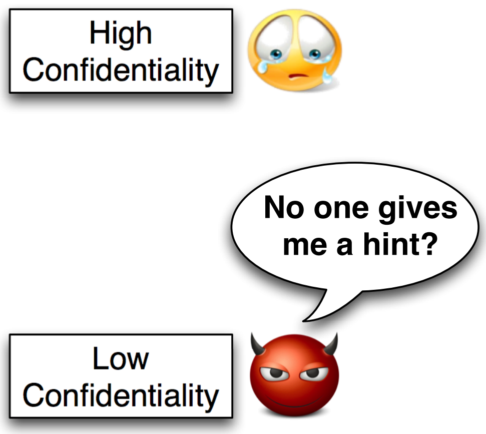
1. There is information flow from a victim's high confidentiality level to an attacker's low confidentiality level;
2. The information flows observable by a low confidentiality subject (e.g., an attacker program) should be unambiguous.

4.2 Cache Model of Side-channel Leakage

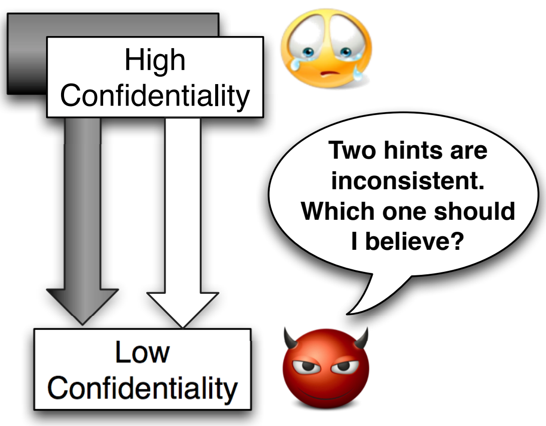
Now let's apply our general model to the side-channel leakage in the cache system. A cache shared between the attacker and the victim can be modeled as a FSM. We define the victim's lines in the cache as the high confidentiality (*Hi*) information, and the attacker's lines in the cache as the low confidentiality (*Lo*) information. For category I cache attacks, the attacker observes cache misses due to external interference. Information flow from *Hi* to *Lo* occurs when the victim's cache lines interfere with the attacker's cache lines. In the next few sections, we build models for the different cache architectures described earlier and propose an *Interference Matrix* to study the information flow in these caches.

4.2.1 Conventional Cache.

We model each cache line in a conventional cache as being in one of three states: **A** (occupied by the attacker - or low confidentiality, *Lo*), **V** (occupied by the victim - or high confidentiality,



(a) No information flow



(b) Ambiguous information flow

Figure 16: No information leakage

Hi) or **I** (invalid - does not have any valid contents). The state machine of a single cache line is shown in figure 17, including the transitions between different states. For clarity of exposition, let's assume that only two programs are running: the attacker and the victim. Then there are four events that cause state transitions: V_miss, the victim has a cache miss for a memory block that maps into this cache line, A_miss, the attacker has a cache miss for data that maps into this cache line, and similarly, V_hit and A_hit, which indicate a cache hit for this cache line by the victim or the attacker, respectively. The state of the entire cache is the combination of all the cache line states.

We define V_i as the victim's cache line with an index of i. This is the high confidentiality information. We define A_i as the attacker's cache line on set i. This is the low confidentiality information. Whenever a cache line occupied by an attacker is over-written (replaced) by a cache line belonging to a victim, we have a violation of the "No write down" rule of BLP. This replacement can be observed by the attacker by detecting a cache miss when he next accesses this cache line that he had previously filled with his own data. This is an information flow from the victim's high confidentiality level to the attacker's low confidentiality level. We denote $A_i \dashrightarrow V_i$ as such a case of a cache miss due to external interference. The reverse situation where a cache line occupied by a victim at high confidentiality is replaced by an attacker at low confidentiality is allowed in the BLP policy, since the high confidential program (the victim) is allowed to read (observe) low confidentiality information. Hence, this reverse situation does not leak high confidentiality information, and is not labelled an "external interference" information leak in figure 17.

Now we consider how to model the transitions in the cache line states. Table 3 shows the transition rules for a conventional cache. It has four columns. With the *Input*, the cache will transit from the *Current State* to the *Next State*, and produce the *Output*. Assume i is the number of ways for set-associativity, and j is the number of cache sets. Then the State Matrix is the current and next state of each line: $S_{p,q} = \{\mathbf{A}, \mathbf{V}, \mathbf{I}\}$ is the state of set p and way q. The Input Vector: $T_p = \{A\_miss, A\_hit, V\_miss, V\_hit\}$ is the cache event on set p that causes a state transition from $S_{p,q}$ to $S'_{p,q}$.

We use a Replacement Matrix: $l_{p,q} = \{0, ..., i-1\}$ to model the LRU (Least Recently Used replacement policy) order of set p and way q to be replaced. If $l_{p,q} = 0$, the line in set p and way q is in the highest priority to be replaced. If $l_{p,q} = i-1$, then this line has the lowest priority to be replaced.

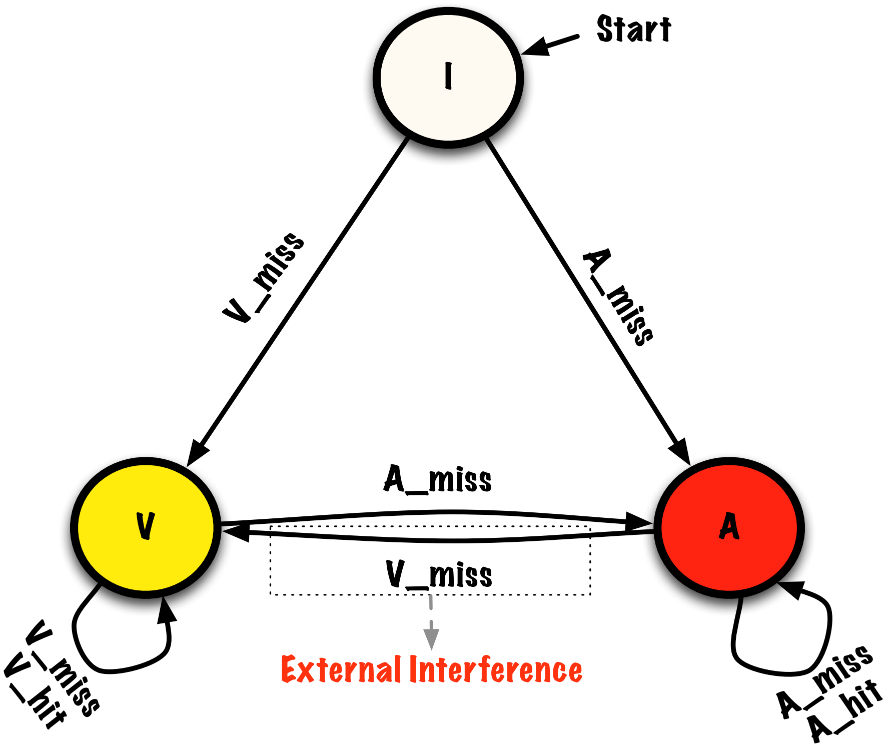


Figure 17: State machine for a single line of a conventional cache.

A particularly novel aspect of our state machine model of a cache is the modeling of the information leak as a pseudo output which we call the Interference Matrix. For this paper, the Interference Matrix: $A_p \rightarrow V_q = \{1, 0\}$ depicts the external interference between two processes which leaks confidential (protected) information, as described above for the BLP security policy model, when an attacker cache line in set p is replaced by a victim cache line with a cache index of q. In conventional caches, shown in Figure 17 and Table 3, a victim process would replace A_q with V_q. However, in randomized cache approaches, V_q may replace a cache line in a randomly selected set p – thus replacing A_p with V_q instead. This will become clearer in the following descriptions of different cache architecture models.

We propose a novel use of Murphi [24, 25] to realize the above model. Murphi is a FSM model checker, used to verify the invariants of the system by enumerating all the explicit states. Instead of checking the invariants, we use Murphi to go over all the possible cache states and count the number of each kind of external interference. Then, we generate the Interference Matrix (IM) by calculating the ratio of each type of external interference as:

$$IM(A_i, V_j) = \frac{\text{\# of external interferences } (V_j \rightarrow A_i)}{\text{total \# of external interferences}} \tag{2}$$

Without loss of generality, we assume a 3-set, 2-way set-associative conventional cache. We consider 10 rounds of memory accesses. The results are shown in table 4. From this table, we can see that for a conventional cache, (1) there is information flow from V_i to A_i, and (2) each A_i is only replaced (interfered) by V_i, so the interferences at different times are quite unambiguous. Since it satisfies both criteria for leakiness, we conclude *the conventional cache is very leaky.*

Table 3: Conventional cache state transition

Current State	Input	Output	Next State
State Matrix	**Input Vector**	**Interference Matrix**	**State Matrix**
$\begin{bmatrix} S_{0,0} & \cdots & S_{0,i-1} \\ \vdots & \ddots & \vdots \\ S_{j-1,0} & \cdots & S_{j-1,i-1} \end{bmatrix}$	$\begin{bmatrix} T_0 \\ \vdots \\ T_{j-1} \end{bmatrix}$	$\begin{bmatrix} A_0 \rightarrow V_0 & \cdots & A_0 \rightarrow V_{j-1} \\ \vdots & \ddots & \vdots \\ A_{j-1} \rightarrow V_0 & \cdots & A_{j-1} \rightarrow V_{j-1} \end{bmatrix}$	$\begin{bmatrix} S'_{0,0} & \cdots & S'_{0,i-1} \\ \vdots & \ddots & \vdots \\ S'_{j-1,0} & \cdots & S'_{j-1,i-1} \end{bmatrix}$
Replacement Matrix			**Replacement Matrix**
$\begin{bmatrix} l_{0,0} & \cdots & l_{0,i-1} \\ \vdots & \ddots & \vdots \\ l_{j-1,0} & \cdots & l_{j-1,i-1} \end{bmatrix}$			$\begin{bmatrix} l'_{0,0} & \cdots & l'_{0,i-1} \\ \vdots & \ddots & \vdots \\ l'_{j-1,0} & \cdots & l'_{j-1,i-1} \end{bmatrix}$

Table 4: Interference Matrix for conventional cache

Set ID	V_1	V_2	V_3
A_1	33.3%	0.0%	0.0%
A_2	0.0%	33.3%	0.0%
A_3	0.0%	0.0%	33.3%

(total # of interferences: 27,996)

4.2.2 Static-Partitioning Cache.

SP cache is modeled in figure 18. The difference with the conventional cache is that each cache line can only have two states (**I** and **V**, or **I** and **A**). So transitions of $A_j \rightarrow V_i$ or $V_i \rightarrow A_j$ can never happen. The Interference Matrix of an SP cache is shown in Table 5. We can clearly see that there is no interference between A_j and V_i for SP cache. So there is no information flow from any V_i to any A_j. It does not satisfy the first criterion of side-channel leakage. So *SP cache can effectively reduce this category of side-channel leakage to zero.*

Table 5: Interference Matrix for SP cache

Set ID	V_1	V_2	V_3
A_1	0.0%	0.0%	0.0%
A_2	0.0%	0.0%	0.0%
A_3	0.0%	0.0%	0.0%

(total # of interferences: 0)

4.2.3 Partition-Locked Cache.

We consider two uses of PL cache: without preload and with preload. For PL cache without preload (figure 19a), the cache is initially empty. Both the victim and attacker can fill the cache

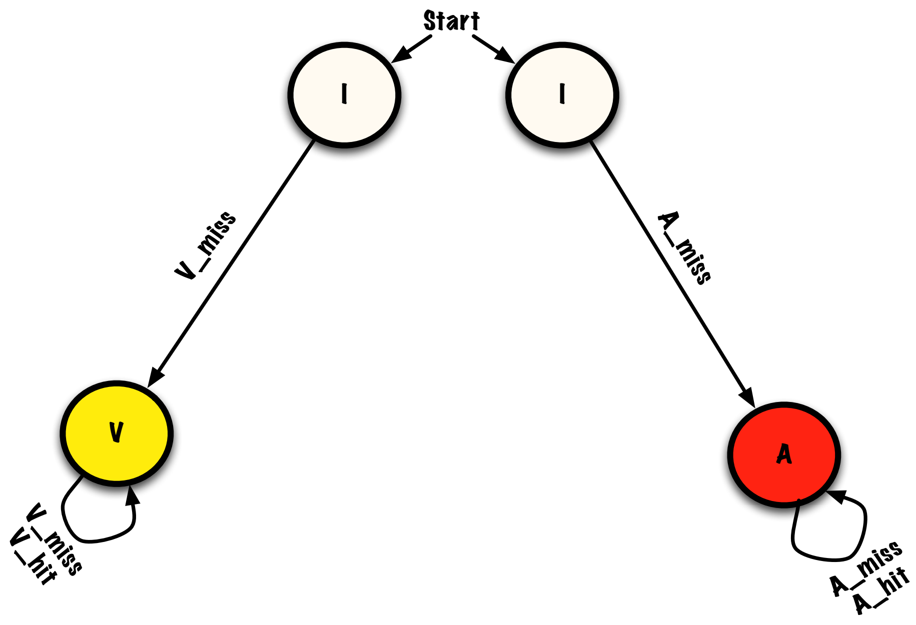


Figure 18: State machine for a single line of SP cache.

Table 6: Interference Matrix for PL cache

(a) without preload

Set ID	V_1	V_2	V_3
A_1	33.3%	0.0%	0.0%
A_2	0.0%	33.3%	0.0%
A_3	0.0%	0.0%	33.3%

(total # of interferences: 13,794)

(b) with preload

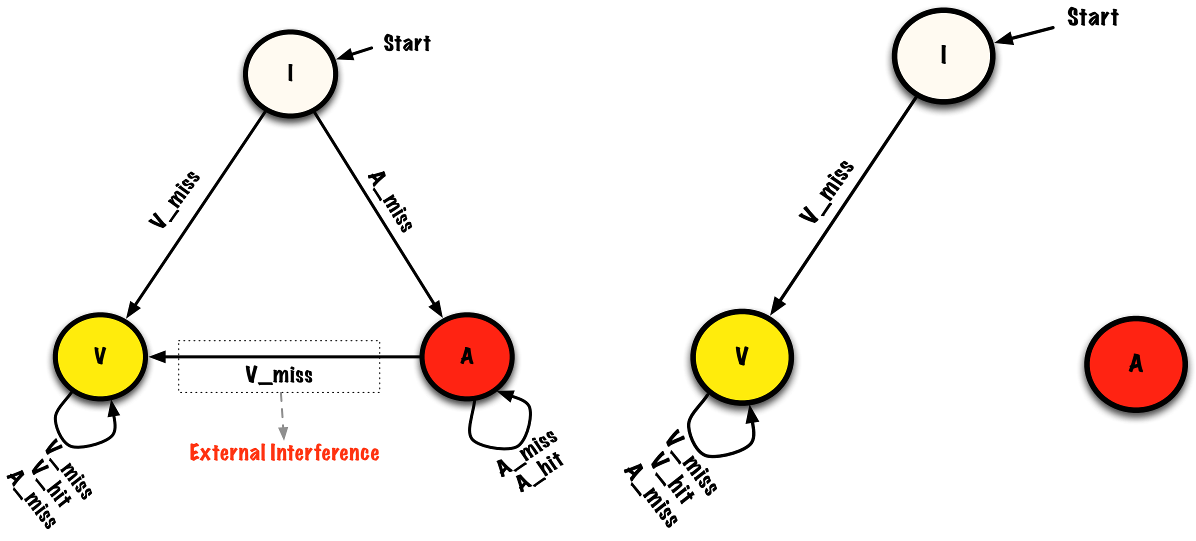
Set ID	V_1	V_2	V_3
A_1	0.0%	0.0%	0.0%
A_2	0.0%	0.0%	0.0%
A_3	0.0%	0.0%	0.0%

(total # of interferences: 0)

with its data. However, once the victim's cache lines are locked in the cache, they can not be replaced by the attacker. So we have the transition of $A_i \to V_i$, but $V_i \to A_i$ is forbidden. For PL cache with preload (figure 19b), the victim initially occupies the cache line and locks it in the cache. So neither transition of $A_i \to V_i$, or $V_i \to A_i$ can happen. Tables 6a and 6b display the Interference Matrix of PL cache without and with preload. PL cache without preload has the same interference distribution as the conventional cache, which indicates that *PL cache, without preloading can leak information when loading the victim's cache lines into the cache for the first time.* PL cache with preload has the same interference distribution as the SP cache, which indicates that *with proper usage like preloading the victim's sensitive cache lines, PL cache prevents information leakage.* So even through the PL cache without preload can survive the "Prime and Probe" attack, the abstract model still reveals its vulnerability: there may be information leakage targeting the cache warm-up stage.

4.2.4 Random-Eviction Cache.

Now let us consider the randomization approach. The state machine of one line for a Random-Eviction cache is shown in figure 20. Comparing with a conventional cache, here we add two



(a) PL cache without preload (b) PL cache with preload

Figure 19: State machine for a single line of PL cache

Table 7: Random-Eviction cache state transition

Current State	Input	Output	Next State
State Matrix	**Input Vector**	**Interference Matrix**	**State Matrix**
$\begin{bmatrix} S_{0,0} & \cdots & S_{0,i-1} \\ \vdots & \ddots & \vdots \\ S_{j-1,0} & \cdots & S_{j-1,i-1} \end{bmatrix}$	$\begin{bmatrix} T_0 \\ \vdots \\ T_{j-1} \end{bmatrix}$	$\begin{bmatrix} A_0 \rightarrow V_0 & \cdots & A_0 \rightarrow V_{j-1} & A_0 \rightarrow I \\ \vdots & \ddots & \vdots & \vdots \\ A_{j-1} \rightarrow V_0 & \cdots & A_{j-1} \rightarrow V_{j-1} & A_{j-1} \rightarrow I \end{bmatrix}$	$\begin{bmatrix} S'_{0,0} & \cdots & S'_{0,i-1} \\ \vdots & \ddots & \vdots \\ S'_{j-1,0} & \cdots & S'_{j-1,i-1} \end{bmatrix}$
Replacement Matrix	**Eviction Vector**		**Replacement Matrix**
$\begin{bmatrix} l_{0,0} & \cdots & l_{0,i-1} \\ \vdots & \ddots & \vdots \\ l_{j-1,0} & \cdots & l_{j-1,i-1} \end{bmatrix}$	$\begin{bmatrix} E_0 \\ \vdots \\ E_{j-1} \end{bmatrix}$		$\begin{bmatrix} l'_{0,0} & \cdots & l'_{0,i-1} \\ \vdots & \ddots & \vdots \\ l'_{j-1,0} & \cdots & l'_{j-1,i-1} \end{bmatrix}$

more transitions: $\mathbf{A} \rightarrow \mathbf{I}$ (attacker's line is randomly chosen to be evicted) and $\mathbf{V} \rightarrow \mathbf{I}$ (victim's line is randomly chosen to be evicted). Among them, $\mathbf{A} \rightarrow \mathbf{I}$ is introduced as a new interference. This interference will cause the attacker's observation of cache miss, but it is not due to the victim's execution. So we call it *fake interference*. The existence of fake interference can help to reduce the information leakage as it can satisfy the two criteria we proposed: (1) There is no information flow from the victim since it does not carry any victim information. (2) It makes the attacker's observation ambiguous as the attacker cannot differentiate the cache miss due to the victim's execution or to random invalidation. Table 7 gives the transition rules of RE cache. Comparing with the conventional cache, we add an Eviction Vector: $E_p = \{0, ..., i\}$ to indicate which cache line in set p is to be evicted. We also add a new column in the Interference Matrix to denote the fake interference: $A_p \rightarrow I = \{1, 0\}$

The Interference Matrix of RE cache is shown in Table 8. The results show that a large amount of interference is fake (23.6%). Although it is still possible for the attacker to retrieve side-channel information from the rest of the interferences (9.7%), it will be a hard job to filter the noise due to fake interference from the observations. The larger the proportion of fake interference, the more difficult it is for the attacker to retrieve useful information.

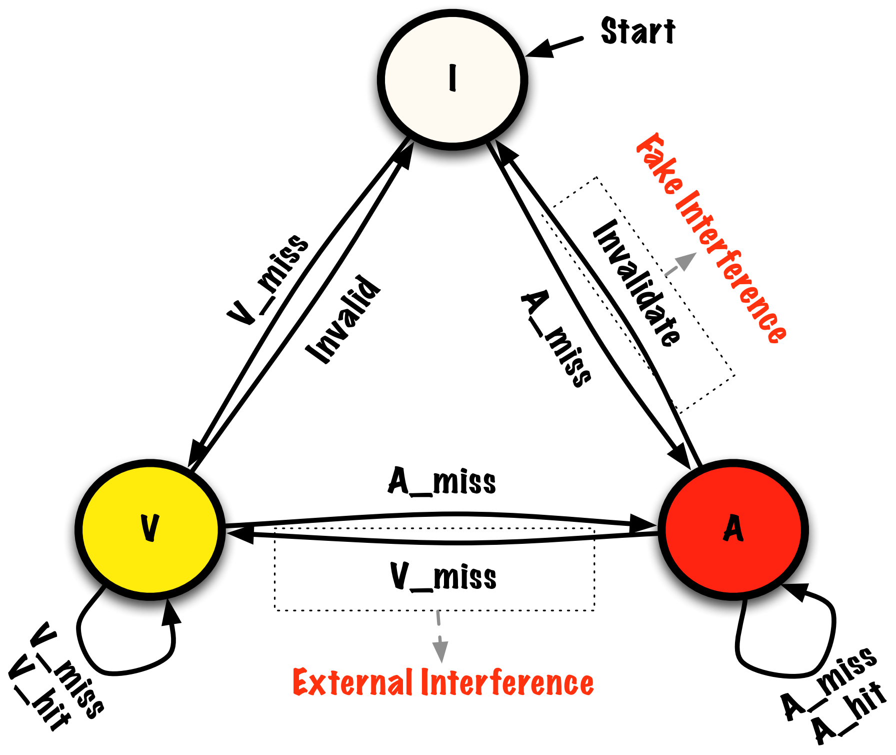


Figure 20: State machine for a single line of RE cache.

Table 8: Interference Matrix for RE cache

Set ID	V_1	V_2	V_3	Fake
A_1	9.7%	0.0%	0.0%	23.6%
A_2	0.0%	9.7%	0.0%	23.6%
A_3	0.0%	0.0%	9.7%	23.6%

(total # of interferences: 117,349,797)

4.2.5 Random-Permutation Cache.

Figure 21 shows the cache line state machine of RP cache. The mapping update procedure involves two sets. When Line ① in set S is in state **V** and encounters an *A_miss*, this line will still stay in state **V**. Instead, a random Line ② in a random set S' is selected. Whatever state Line ② is in, it will be replaced by the incoming attacker's line and jump to state **A**. All the lines of set S and S' that are in state **A** will be evicted out of the cache (Line ③ in figure 21). In the meantime, the mappings of set S and S' will be swapped in the attacker's *permutation table*. A similar procedure happens when Line ① is in state **A** and encounters a *V_miss*. Line ④ in figure 21 is randomly selected and replaced, and Line ⑤ is evicted out of the cache when swapping the two sets in the victim's *permutation table*.

Table 9 shows the transition rules of RP cache. Comparing with RE cache, we add the victim's *permutation table* $V_p = \{0, ..., j-1\}$ and the attacker's *permutation table* $A_p = \{0, ..., j-1\}$ in the Current State, Input as well as the Next State.

Table 10 displays the Interference Matrix for RP cache. We can see the fake interference constitutes a rather high percentage of the total interferences (26.8 %). This is due to the line

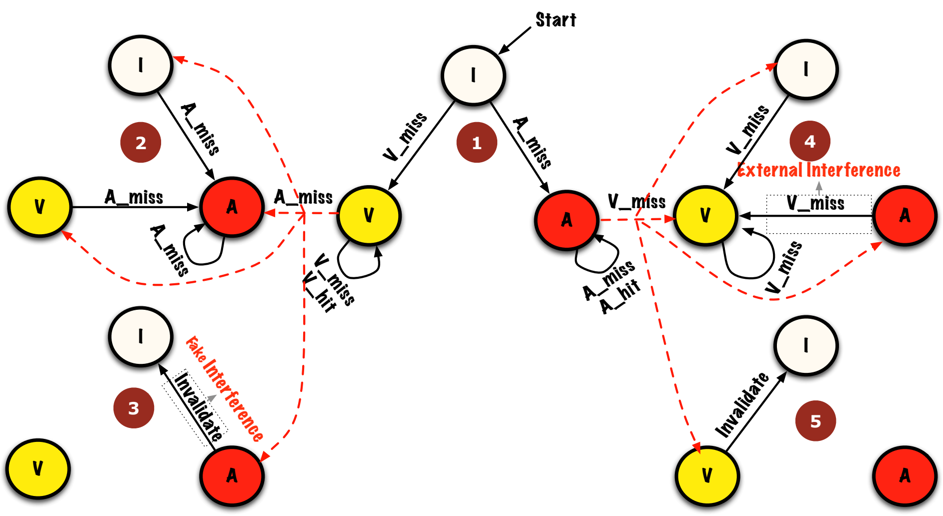


Figure 21: State machine for RP cache.

Table 9: Random-Permutation cache state transition

Current State	Input	Output	Next State
State Matrix	**Input Vector**	**Interference Matrix**	**State Matrix**
$\begin{bmatrix} S_{0,0} & \cdots & S_{0,i-1} \\ \vdots & \ddots & \vdots \\ S_{j-1,0} & \cdots & S_{j-1,i-1} \end{bmatrix}$	$\begin{bmatrix} T_0 \\ \vdots \\ T_{j-1} \end{bmatrix}$	$\begin{bmatrix} A_0 \to V_0 & \cdots & A_0 \to V_{j-1} & A_0 \to I \\ \vdots & \ddots & \vdots & \vdots \\ A_{j-1} \to V_0 & \cdots & A_{j-1} \to V_{j-1} & A_{j-1} \to I \end{bmatrix}$	$\begin{bmatrix} S'_{0,0} & \cdots & S'_{0,i-1} \\ \vdots & \ddots & \vdots \\ S'_{j-1,0} & \cdots & S'_{j-1,i-1} \end{bmatrix}$
Permutation Table	**Permutation Table**		**Permutation Table**
$\begin{bmatrix} V_0 \\ \vdots \\ V_{j-1} \end{bmatrix} \begin{bmatrix} A_0 \\ \vdots \\ A_{j-1} \end{bmatrix}$	$\begin{bmatrix} V'_0 \\ \vdots \\ V'_{j-1} \end{bmatrix} \begin{bmatrix} A'_0 \\ \vdots \\ A'_{j-1} \end{bmatrix}$		$\begin{bmatrix} V'_0 \\ \vdots \\ V'_{j-1} \end{bmatrix} \begin{bmatrix} A'_0 \\ \vdots \\ A'_{j-1} \end{bmatrix}$
Replacement Matrix	**Eviction Vector**		**Replacement Matrix**
$\begin{bmatrix} l_{0,0} & \cdots & l_{0,i-1} \\ \vdots & \ddots & \vdots \\ l_{j-1,0} & \cdots & l_{j-1,i-1} \end{bmatrix}$	$\begin{bmatrix} E_0 \\ \vdots \\ E_{j-1} \end{bmatrix}$		$\begin{bmatrix} l'_{0,0} & \cdots & l'_{0,i-1} \\ \vdots & \ddots & \vdots \\ l'_{j-1,0} & \cdots & l'_{j-1,i-1} \end{bmatrix}$

invalidation when updating the *permutation table*. In addition, we can see there are not huge differences for the interference of each V_i on each A_j, making the attacker's observations quite ambiguous. When the attacker observes a cache miss in set j, it is hard for him to tell which set is accessed by the victim. This is due to the random mapping from memory address to cache set. *Both reasons can enhance the security of RP cache against side-channel leakage.*

4.2.6 NewCache.

Finally, we model NewCache. Figure 22 displays the state machine model of each cache line in NewCache. According to NewCache's replacement policy [11, 19], if there is an index miss for the attacker or the victim, a random cache line (Line ① in figure 22) is selected to be replaced, and Line ① will jump to state **A** (attacker's index miss) or state **V** (victim's index miss). The importance with a conventional cache is that for this cache line p being modeled, the A_miss and V_miss can be from any cache line q, for $q = 1, 2, .., j$, and not just from cache line p. If there is a tag miss for the attacker or the victim for Line ②, this cache line will be directly replaced by the incoming line, jumping from state **A** to **A** (attacker's tag miss) or from state **V** to **V** (victim's

Table 10: Interference Matrix for RP cache

Set ID	V_1	V_2	V_3	Fake
A_1	2.17%	2.19%	2.19%	26.8%
A_2	2.19%	2.17%	2.19%	26.8%
A_3	2.19%	2.19%	2.17%	26.8%

(total # of interferences: 7,842,324)

Table 11: NewCache state transition

Current State	Input	Output	Next State
State Matrix	**Input Vector**	**Interference Matrix**	**State Matrix**
$\begin{bmatrix} S_0 \\ \vdots \\ S_{j-1} \end{bmatrix}$	$\begin{bmatrix} T_0 \\ \vdots \\ T_{j-1} \end{bmatrix}$	$\begin{bmatrix} A_0 \rightarrow V_0 & \cdots & A_0 \rightarrow V_{i-1} & A_0 \rightarrow I \\ \vdots & \ddots & \vdots & \vdots \\ A_{i-1} \rightarrow V_0 & \cdots & A_{i-1} \rightarrow V_{i-1} & A_{i-1} \rightarrow I \end{bmatrix}$	$\begin{bmatrix} S'_0 \\ \vdots \\ S'_{j-1} \end{bmatrix}$
LNreg	**LNreg**		**LNreg**
$\begin{bmatrix} N_0 \\ \vdots \\ N_{j-1} \end{bmatrix}$	$\begin{bmatrix} N'_0 \\ \vdots \\ N'_{j-1} \end{bmatrix}$		$\begin{bmatrix} N'_0 \\ \vdots \\ N'_{j-1} \end{bmatrix}$

tag miss).

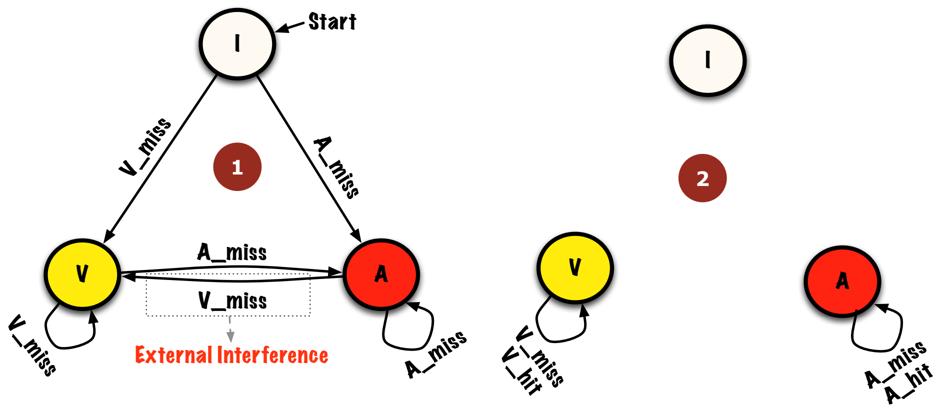


Figure 22: State machine for NewCache.

Table 11 shows the transition rules for NewCache. Assume the number of physical cache lines is j, and the number of LDM cache lines is i, $(i > j)$. We also have the Line Number register (LNreg) $N_p = (\{0, ..., i-1\}, \{\mathbf{A}, \mathbf{V}, \mathbf{I}\})$, recording which lines in the LDM cache are stored in the set p.

We use Murphi to simulate a NewCache with 3 logical cache lines and 2 physical cache lines. The Interference Matrix is shown in table 12. Due to the fully-associative mappings from the LDM to physical cache, the interference of V_p on any A_q happens with the same probability. So the attacker's observations are quite ambiguous. *This means NewCache does not leak information through side-channel attacks due to cache misses arising from external interference.*

Table 12: Interference Matrix for NewCache

Set ID	V_1	V_2	V_3	Fake
A_1	11.1%	11.1%	11.1%	0.0%
A_2	11.1%	11.1%	11.1%	0.0%
A_3	11.1%	11.1%	11.1%	0.0%

(total # of interferences: 1,368,954)

Table 13: Results comparison for different methods

Methods	**Isolation**	**Randomization**
Actual Attack	Conv > {PL, SP}	{Conv, RE1000} > RE10 > {RP, New}
Cache Model	{Conv, PL-w/o preload} > {PL-w/ preload, SP}	Conv > RE > RP > New

4.3 Discussion

4.3.1 Cache Defenses.

In section 4.1, we use information flow and the BLP policy to build abstract models, which can show the information leakage. Specifically, we propose two criteria that can cause side-channel information leakage: (1) there is information flow from a victim's high confidential level to an attacker's low confidential level; (2) the side-channel information the attacker observes is unambiguous at different times, so the attacker can get correct conclusions through the off-line statistical analysis. When we use this model to evaluate the cache architectures, it is interesting to find that different approaches usually focus on different criteria. For the isolation approach, the defenses usually try to nullify the first criterion: eliminating the information flow to reduce the information leakage. From the Interference Matrix of SP and PL (with preload) cache, the total number of interferences between the victim and the attacker is zero, so there is no information flow. For the randomization approach, the defenses usually try to nullify the second criterion: using randomization to increase the inconsistency of the attacker's observations. From the Interference Matrix of RE, RP and NewCache, each A_i observed by the attacker can be caused by any V_j of the victim's action with approximately equal probabilities, or caused by the fake interference, so the ambiguous interferences create difficulties for the attacker to get accurate conclusions. Either way is effective at reducing the side-channel leakage. The elimination of one criterion or both can inspire researchers to propose more defense approaches other than isolation and randomization.

4.3.2 Extensions.

In our cache model, we count the cache misses that will cause external interference between the attacker and the victim. This can help to evaluate category I cache attacks. However, with some modifications, our model can also be applied to the other three categories in Table 1. For instance, if we want to model category III cache attacks induced by cache misses due to internal interference, we can consider only transitions of "*V_miss*" from state **V** to **V** and create such an Interference Matrix. If we want to model category IV attacks induced by cache hits due to internal interference, we can count the transitions of "*V_hit*" from state **V** to **V** and create such an Interference Matrix. If we are interested in category II attacks induced by cache hits due to external interference, the cache line needs a fourth state **A/V**, which means this line contains the shared memory between the victim and the attacker. Thus we can count the transitions of "A/V_hit" from state **A/V** to **A/V** and create the corresponding Interference Matrix. So our cache models are very flexible and can be applied to model different types of attacks.

5 Comparisons of Evaluation Methods

We compare the features of these two evaluation methods. Running the actual attack program on the caches with the off-line key recovery is more realistic, while the Interference Matrix is only an abstract model.

For scope, the attack program only evaluates the "Prime and Probe" attack on AES, so it is cipher-specific and has a very limited scope. The cache models with Interference Matrix consider all the attacks induced by external interference due to cache misses. It applies to all attacks in category I and has a much broader scope.

Table 13 shows that the conclusions we get from the two evaluation methods are consistent. For the isolation approach, PL and SP caches can effectively defend against side-channel attacks. For the randomization approach, RP and NewCache are also very effective in reducing side-channel leakage. RE caches are attackable, but if the eviction frequency increases, the attack will become harder.

6 Related Work

For actual attack programs, researchers have designed different side-channel attacks [13, 14, 15, 4, 3] on the caches. But they seldom analyzed the feasibility of these attacks. In [26], Success Rate and Guessing Entropy are defined as general metrics to evaluate the feasibility of side-channel key recovery. Then [27] defines the average Success Rate to evaluate the profiled cache timing attacks. It also builds an analytical model to estimate the Success Rate for determining the best attack strategy. Our proposed key-byte votes is a novel metric.

Some side-channel metrics have also been proposed to accelerate the evaluation processes. In [28], a metric called Side-channel Vulnerability Factor (SVF) was proposed to measure the information leakage. However, SVF has some limitations in its scope, definition and measurements. These issues have been discussed and corrected, and an improved metric called Cache Side-channel Vulnerability (CSV) metric was also proposed in [29]. [30] also made some improvements over SVF and proposed the timing-SVF metric for timing-based cache side-channel attacks. However, the timing-SVF still needs the execution of an actual attack, requiring a large number of samples for accuracy. In [26], the authors applied mutual information theory to evaluate the feasibility of key-recovery, and try to find the connections between mutual information with Success Rate and Guessing Entropy. [31] proposed static analysis to establish formal security guarantees against cache side-channel attacks, which can capture the upper bound of the side-channel information leakage.

Some side-channel models and formal verification methods have also been proposed. [32] built the models of timing side-channel leakage from the program code level. [33] uses the technique of self-composition to do the formal verification of cryptographic software. [34] presented an information-theoretic metric for adaptive side-channel attacks, which can tell the attacker's remaining uncertainty in order to adjust his strategy. These models do not reflect the features of cache architectures and cache-based attacks, so they cannot be applied to evaluate cache side-channel information leakage. Unlike these past models, we are the first to model the cache architecture itself, and define the Interference Matrix, which includes the cache state transitions that can leak information.

7 Conclusions

This work proposes new methods for the evaluation of a system's vulnerability to cache side-channel attacks. We categorize the cache attacks based on their root cause, and focus on the most powerful cache side-channel attacks: category I attacks induced by cache misses due to external interference.

In the first evaluation method we propose, we design and run actual attacks on different caches and recover the keys to evaluate the feasibility of such an attack. We define the number of key-byte votes as a new metric to quantitatively compare the vulnerability of different cache architectures to a given attack. In the second method, we build general models of cache architectures and define the Interference Matrix to reveal the sources of information leakage, and the frequcency of leakage for each source. We compare the features of the methods and their results, and find them to be consistent.

Our evaluation methods can be applied to other categories of cache side-channel attacks. For evaluation with actual attacks, we can run other types of attacks proposed recently. Calculating the votes of candidate keys is still applicable since it is general for all key-recovery methods. Our new models of caches and interferences can be extended to other attacks, as discussed in section 4.3.2. Future work can extend these evaluation methods to new cache architectures, and information leakage from other side channels.

References

1. Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. In *Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, pages 2–21, 1991.
2. Mitsuru Matsui. Linear cryptanalysis method for des cipher. In *Workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 386–397, 1994.
3. David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011.
4. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, pages 1–20, 2006.
5. Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Investigations of power analysis attacks on smartcards. In *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, pages 17–17, 1999.
6. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, 1999.
7. N. Homma, T. Aoki, and A. Satoh. Electromagnetic information leakage for side-channel analysis of cryptographic modules. In *Electromagnetic Compatibility (EMC), 2010 IEEE International Symposium on*, pages 97–102, 2010.
8. Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760, 2004.
9. Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing

attack. In *Proceedings of the The International Conference on Smart Card Research and Applications*, pages 167–182, 2000.

10. Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

11. Zhenghong Wang and R.B. Lee. A novel cache architecture with enhanced performance and security. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 83 –93, nov. 2008.

12. Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. 8(4):35:1–35:21, January 2012.

13. Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

14. Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

15. Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. cryptographic hardware and embedded systems. In *Lecture Notes in Computer Science series 4249*, pages 201–215. Springer, 2006.

16. Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on aes, 2006.

17. Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18, 2007.

18. Zhenghong Wang. *Information Leakage Due to Cache and Processor Architectures*. Phd thesis, Princeton University, Princeton, NJ, 2012.

19. Fangfei Liu and Ruby B. Lee. Security testing of a secure cache design. In *in Hardware and Architectural Support for Security and Privacy (HASP) workshop at ISCA*, 2013.

20. The gem5 Simulator System. http://www.gem5.org.

21. Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

22. David. Elliott Bell and Leonard J. Lapadula. Secure computer systems: Mathematical foundations, 1973.

23. Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley InterScience, 2008.

24. John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using murphi. IEEE Computer Society Press, 1997.

25. David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proc. of the Int. Conference on Computer Design: VLSI in Computer & Processors*, ICCD, Oct. 1992.

26. François-Xavier Standaert, Tal G. Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques*, pages 443–461, 2009.

27. C. Rebeiro and D. Mukhopadhyay. Boosting profiled cache timing attacks with a priori analysis. *Information Forensics and Security, IEEE Transactions on*, 7(6):1900–1905, 2012.

28. John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: a metric for measuring information leakage. In *International Symposium on Computer Architecture*, pages 106 –117, June 2012.

29. Tianwei Zhang, Si Chen, Fangfei Liu, and Ruby B. Lee. Side channel vulnerability metrics: the promise and the pitfalls. In *in Hardware and Architectural Support for Security and Privacy (HASP) workshop at ISCA*, 2013.

30. Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. Hardware prefetchers leak: A revisit of svf for cache-timing attacks. In *in Hardware and Architectural Support for Security and Privacy (HASP) workshop at Micro*, 2012.

31. Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th international conference on Computer Aided Verification*, pages 564–580, 2012.

32. Josef Svenningsson and David Sands. Specification and verification of side channel declassification. In *Proceedings of the 6th international conference on Formal Aspects in Security and Trust*, pages 111–125, 2010.

33. J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.*, 78(7):796–812, July 2013.

34. Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 286–296, 2007.