New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels

Tianwei Zhang
Princeton University
tianweiz@princeton.edu

Ruby B. Lee
Princeton University
rblee@princeton.edu

ABSTRACT

Side-channel attacks try to breach confidentiality and retrieve critical secrets through the side channels. 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. However, there are currently no means for comparing and evaluating the effectiveness of different defense solutions against these attacks.

In this paper, we propose a novel method to evaluate a system's vulnerability to side-channel attacks. We establish side-channel leakage models based on the *non-interference* property. Then we define how the security aspects of a cache architecture can be modeled as a finite-state machine (FSM) with state transitions that cause interference. We use *mutual information* to quantitatively reveal potential side-channel leakage of the architectures, and allow comparison of these architectures for their relative vulnerabilities to side-channel attacks. We use real attacks to validate our results.

1. INTRODUCTION

Confidentiality is a major concern in information security. Strong encryption is often used for confidentiality protection. Many attacks have been designed to break these cryptographically protected systems. Among them, side-channel attacks exploit the physical characteristics of the system to derive the crypto keys. These attacks are serious security threats for several reasons. First, side-channel attacks target the vulnerabilities of the systems instead of the cryptographic algorithms. So the same attack strategies 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 [1]), which may 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. Fourth, side-channels exist widely in different systems. Power [2,3], electromagnetic radiation [4,5], timing [6], etc., can all be exploited by the attacker to infer the inaccessible critical information. Fifth, the observable side-channel information is due to the inherent physical features of the system, so it is very difficult to eliminate these side-channels.

ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664273.

A popular target of side channel attacks is the hardware cache. Caches are one of the most important features for improving performance in modern processors. Their use in the memory hierarchy significantly reduces the memory access time. However, the different access times due to fast cache hits versus slow cache misses can be exploited in cache side channel attacks to leak information. [1,7–9]. It is typically unacceptable to eliminate this side channel by disabling the cache, because of the severe performance degradation that would result. Recent work has also shown the possibility of cross-VM side-channel attacks in cloud computing [10], making side-channel attacks a real and serious threat.

To defend against cache side-channel attacks, a variety of software defenses [1,8,11,12] have been proposed. These defenses tend to degrade performance severely [11], and are not applicable to arbitrary programs [12]. Furthermore, software defenses are not completely secure, since software has no control over hardware caches. For example, a software solution [1,8,11] may add instructions to load an entire security-sensitive table into the cache, before each access to this table, in the software-implemented cipher. However, this does not prevent the cipher from being context-switched out and having the cache lines containing table entries evicted by another program (possibly the attacker). Also, hardware cache controllers have the freedom to invalidate, evict or replace any cache lines – beyond the control of any software.

Hardware defenses have also been proposed to mitigate information leakage through cache side-channels, with the goal of providing more security and higher performance than software defenses. The idea is to rethink cache architectures to thwart certain types of side-channel attacks without impacting the essential performance provided by caches. Different secure cache architectures have been proposed [13–15]. The performance of these architectures can be tested by performance benchmarks, but their security effectiveness have only been analyzed qualitatively. Currently, there are no reliable methods for comparing the effectiveness of different secure cache architecture approaches. In this paper, we propose a new way of modeling caches to answer two questions: (1) *Do these secure caches really defend against cache side-channel attacks?* and (2) *What are the relative vulnerabilities of different cache architectures to different side-channel attacks?*

Our approach is to model the security aspects of different cache architectures as finite state machines, and accurately model the *interference* property which can show preservation or breaches of confidentiality under side-channel attacks. Our cache state machines include different *subjects* and *states*. Transitions between different *states* can leak confidential information between different *subjects*, when the *non-interference* property is violated. In addition, we propose some quantitative representations of the potential vulnerabilities of the modeled cache architectures to the side channel attacks. To the best of our knowledge, our method of modeling the security aspects of cache architectures and interferences is novel, as are our quantitative measures of side-channel vulnerability which allow comparison of different cache defense strategies and analysis of the root causes of side-channel leakage.

The key contributions of this paper are:

- Showing how to apply the principle of *non-interference* to the modeling of side-channel information leakage;
- Exploiting *mutual information* for measuring the side-channel information leakage, and identifying three conditions for achieving *non-interference* properties;
- A new way of building finite-state machines for modeling the security aspects of cache architectures, and using a model-checking tool for quantitative characterization of the systems' side-channel vulnerabilities;
- Verifying our cache security models and their relative side-channel vulnerabilities with real attacks.

The rest of the paper is organized as follows: Section 2 gives the background of cache side-channel attacks and different secure cache defenses. Section 3 describes our side-channel leakage models and quantification for measuring information leakage. In Section 4, we show how to build new security models of cache architectures. In Section 5, we integrate the side-channel leakage model into the cache security models and quantify the caches' potential information leakage. We consider other side-channel attacks in Section 6. Section 7 uses experimental data with real attacks to validate our security models of the different cache architectures. Section 8 discusses related work. Section 9 gives our conclusions and suggestions for future work.

2. BACKGROUND

2.1 Cache Side-channel Attacks

Cache side-channel attacks are particularly dangerous as they are simple software attacks on essential hardware components. The attacker does not need to share the address space with the victim and access its memory. However, he shares the hardware caches with the victim, which provides the attacker a side channel to observe the victim's secret information: a victim's programs executing on the system may have different cache behaviors (hits or misses) when memory accesses are made. These behaviors have different timing characteristics. The attacker tries to capture these characteristics, and deduce the victim's memory accesses that might help him recover the key and break the ciphers.

A large number of cache side-channel attacks have been proposed during the past few years [1,7–9,16–18]. The root cause of all the existing 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 [13]. Combined with the cache behaviors the attackers want to observe (cache misses or hits), we have four cache side-channel attack categories [19] shown in Table 1 and described below.

Table 1: Cache side-channel attack categories

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

Type I: Attacks based on Cache Misses due to External Interference. In this class of attacks, the attacker and the victim 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 cache attacks belong to this class, and a typical one is Percival's attack [7].

Type II: Attacks based on 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; this can give the attacker information about the victim's memory accesses. Some timing-based cache attacks belong to this class, such as Bernstein's attack [8].

Type III: Attacks based on 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, or some, shared memory blocks out of the cache. After a certain time interval of the victim's execution, the attacker reads the shared memory blocks and measures the access time. A short 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 [1] belongs to this class.

Type IV: Attacks based on Cache Hits due to Internal Interference. Similar to type II attacks, 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 the "cache collision" (i.e., cache hits) of memory accesses. Some timing-based attacks belong to this class, such as Bonneau's attack [9].

2.2 Cache Defenses and Architectures

Different cache defenses and secure cache architectures have been proposed to protect against cache side-channel attacks. Basically these designs follow one of two strategies: *Partitioning* or *Randomization* [20]:

2.2.1 *Partitioning.*

Caches can be exploited as side-channels in the external interference attacks because the attacker and the victim can share the caches, and thus interfere with each other's cache usage. So one straightforward approach to prevent information leakage is to prevent the cache sharing 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 (like columns) or by sets (like rows). 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 well-known cost of degrading the computer's performance because of the static cache partitions.

Partition-Locked (PL) cache: PL cache [13] uses a finer-grained dynamic cache partitioning policy. In PL cache, each memory line has a protection bit to represent if it needs to be locked in the cache. Once the protected line (e.g., the victim's critical data) is locked in the cache, it can not be replaced by an unprotected line (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 to security-critical data lines (e.g., those containing AES tables). This leverages, for cache security, the Lock-bit already provided by some caches to improve cache performance for frequently accessed data. The proper use of a PL cache is to preload the sensitive cache lines (e.g., AES table) before encryption begins.

2.2.2 *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 to 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 policy. This will increase the attacker's difficulty in recovering secret information like a cipher key.

Random-Permutation (RP) cache: RP cache [13] uses random memory-to-cache mappings to defend against side-channel attacks. There is a *permutation table* 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 line *D* into the cache, it checks **A**'s *permutation table* and finds the corresponding cache line *R* in set *S*. If this *R* belongs to another process **B**, instead of evicting *R*, thus revealing information to outsiders, a random line *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 lines in these two sets belonging to **A** are invalidated. Since process **A**'s memory-to-cache mappings are dynamic, random and unknown to process **B**, process **B** cannot tell which memory addresses process **A** 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 [14,21] randomizes the memory-to-cache mappings by introducing the concept of 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. The mapping from the LDM cache to the physical cache is fully-associative and is realized using *Line Number Registers* (LNreg's). This dynamic and random mapping enhances the security against information leakage, as each memory line can be mapped to any physical cache line with equal probability; and the cache access pattern changes with each execution of the same program. Furthermore, the performance is enhanced since the LDM cache can be much larger than the physical cache, by merely adding extra index bits to each LNreg [14]. For the replacement policy, if the incoming line *D* cannot find any line in the physical cache with the same index (called an index miss), it will randomly choose a line *R* to replace. If the incoming line can find a line *R* in the physical cache with the same index, but the tag of the line (i.e. the rest of the memory address minus the index bits) is different (called a tag miss), then *D* may replace *R* [21]. The advantage of NewCache is achieving the security benefits of dynamic randomized mapping of memory lines to cache lines, with the same (or even better) performance as conventional caches [14].

3. SIDE-CHANNEL LEAKAGE MODELING

We now build a model for side-channel leakage. First we consider a general system which can be modeled as a finite state machine. The state machine consists of a set of *subjects* and *states*. Each *subject* may provide some *actions* to the machine, causing it to transition from one *state* to another, and generate some *observations*. In order to study the information flow between different *subjects* in this machine, we use the concept of *non-interference* [22], as defined below:

Definition: *Given a state machine $\mathcal{M}$, and its subjects $\mathcal{S}$ and $\mathcal{S}'$, we say $\mathcal{S}$ does not interfere with (or is non-interfering with) $\mathcal{S}'$, if the actions of $\mathcal{S}$ on $\mathcal{M}$ do not affect the observations of $\mathcal{S}'$.*

Then we have the following principle to judge the information flow in the state machine:

Principle: *Given a state machine $\mathcal{M}$, and its subjects $\mathcal{S}$ and $\mathcal{S}'$, if $\mathcal{S}$ is non-interfering with $\mathcal{S}'$, then there is no information flow from $\mathcal{S}$ to $\mathcal{S}'$.*

We now consider the side-channel information leakage. We treat a side channel $\mathcal{C}$ as a state machine. It is a connection between two *subjects*: a victim who performs some *actions* as inputs $\mathcal{I}$ to one side of the channel, and an attacker who retrieves certain *observations* as outputs $\mathcal{O}$ from the other side of the channel. The input $\mathcal{I}$ can change $\mathcal{C}$'s *state*, and affect the output $\mathcal{O}$. So the measurement of side-channel information leakage is equivalent to the evaluation of the *non-interfering* property between the channel's input and output: if $\mathcal{I}$ is non-interfering with $\mathcal{O}$, then there is no side-channel information leakage through $\mathcal{C}$.

Quantification: Side-channel leakage is a statistical process, so we use *mutual information* [23] to quantify the *non-interference* property between the channel's input and output. We denote $P_{\mathcal{I}}(i)$ as the probability that the side channel $\mathcal{C}$ is fed with the input i, $P_{\mathcal{O}}(o)$ as the probability that $\mathcal{C}$ produces the output o, $P_{\mathcal{I},\mathcal{O}}(i,o)$ as the joint probability that $\mathcal{C}$ produces the output o with the input i, $P_{\mathcal{O}|\mathcal{I}}(o|i)$ as the conditional probability that $\mathcal{C}$ produces the output o given the input i, and $P_{\mathcal{I}|\mathcal{O}}(i|o)$ as the conditional probability that the input of $\mathcal{C}$ is i given the output is o. Then the mutual information between $\mathcal{I}$ and $\mathcal{O}$ is defined below:

$$I(\mathcal{I};\mathcal{O}) = \sum_{i\in\mathcal{I}}\sum_{o\in\mathcal{O}} P_{\mathcal{I},\mathcal{O}}(i,o)\ log\left(\frac{P_{\mathcal{I},\mathcal{O}}(i,o)}{P_{\mathcal{I}}(i)P_{\mathcal{O}}(o)}\right) \quad (1a)$$

$$= \sum_{i\in\mathcal{I}}\sum_{o\in\mathcal{O}} P_{\mathcal{O}}(o)\ P_{\mathcal{I}|\mathcal{O}}(i|o)\ log\left(\frac{P_{\mathcal{I}|\mathcal{O}}(i|o)}{P_{\mathcal{I}}(i)}\right) \quad (1b)$$

$I(\mathcal{I};\mathcal{O})$ measures how much information about $\mathcal{I}$ is leaked to $\mathcal{O}$ through the side channel. If $I(\mathcal{I};\mathcal{O})$ is close to zero, then $\mathcal{I}$ is non-interfering with $\mathcal{O}$, and there is little side-channel information flow from $\mathcal{I}$ to $\mathcal{O}$.

Equation 1(b) is the product of three terms. To reduce the mutual information to zero, we can make any of the three terms equal zero. So we have three conditions that can realize the *non-interference* property:

C1 (Output Elimination) *The channel $\mathcal{C}$ does not produce any output. Then the input does not interfere with the output.* We have the following expression:

$$\forall o \in \mathcal{O},\ \ P_{\mathcal{O}}(o) \approx 0 \quad (2)$$

C2 (Noise Domination) *For any output o, if its generation is due to the channel's inherent noise instead of the input, then the output is not affected by the input:*

$$\forall o \in \mathcal{O}, i \in \mathcal{I},\ \ P_{\mathcal{I}|\mathcal{O}}(i|o) \approx 0 \quad (3)$$

C3 (Input Ambiguity) *For any output o, the input of the channel $\mathcal{C}$ can be any i with the same probability. Then the input does not interfere with the output.* That is:

$$\forall i \in \mathcal{I}, o \in \mathcal{O},\ \ P_{\mathcal{I}|\mathcal{O}}(i|o) \approx const$$

Plugging this into the probability equation gives

$$P_{\mathcal{I}}(i) = \sum_{o'\in\mathcal{O}} P_{\mathcal{I},\mathcal{O}}(i,o') = \sum_{o'\in\mathcal{O}} P_{\mathcal{I}|\mathcal{O}}(i|o')P_{\mathcal{O}}(o')$$
$$\approx P_{\mathcal{I}|\mathcal{O}}(i|o)\sum_{o'\in\mathcal{O}} P_{\mathcal{O}}(o') = P_{\mathcal{I}|\mathcal{O}}(i|o)$$

This gives the desired expression of Input Ambiguity:

$$\forall o \in \mathcal{O}, i \in \mathcal{I},\ \ log\left(\frac{P_{\mathcal{I}|\mathcal{O}}(i|o)}{P_{\mathcal{I}}(i)}\right) \approx 0 \quad (4)$$

These three conditions are the key ideas for designing countermeasures to mitigate side-channel leakage. In Section 5, we show how to use these conditions to evaluate the side-channel leakage from different cache architectures.

4. SECURITY MODELING OF HARDWARE CACHE ARCHITECTURES

In this section, we show how to build finite state machines to model the security aspects of cache architectures.

4.1 Cache State Machine

We first consider the state machine of a single cache line shared by the attacker and the victim, which is shown in Figure 1. Each cache line can be in one of three states: **A** (occupied by the attacker), **V** (occupied by the victim) or **INV** (invalid - does not have any valid contents). There are five events that cause state transitions: *V_miss*, the victim has a cache miss for a memory line 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; and *Invalidate*, the cache clears out the data to invalidate this line.

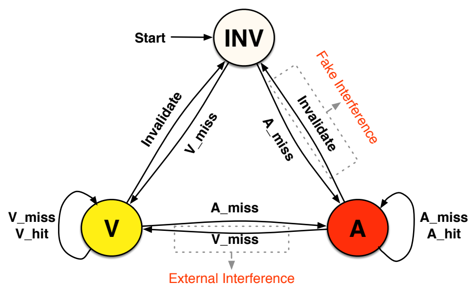


Figure 1: State machine for a single cache line

Now we consider how to model the whole cache as the combination of all the cache lines, and the dynamic operations within the cache state machine. Table 2 shows our model structure and transition rules for the cache states. It has four columns. For each **Event**, the cache will transition from the **Current State** to the **Next State**, and output the **Information Flow Log**. Assume m is the number of ways for set-associativity (columns), and n is the number of cache sets (rows). Then:

Current/Next State: *State Matrix* $S_{p,q} = \{\mathbf{A}, \mathbf{V}, \mathbf{INV}\}$ is the state of the cache line for set p and way q. *Replacement Matrix* $l_{p,q} = \{0, ..., m-1\}$ is used to model the LRU (Least Recently Used) replacement policy ordering of set p and way q for replacement, when a new line has to be brought into the cache set. If $l_{p,q} = 0$, the line in set p and way q has the highest priority to be replaced. If $l_{p,q} = m-1$, then this line has the lowest priority to be replaced in the set.

Event: We define the *Event Vector* as $T_p = \{A\_miss, A\_hit, V\_miss, V\_hit, Invalidate\}$, which is the cache action on set p that causes a state transition from $S_{p,q}$ to $S'_{p,q}$, following the rules in Figure 1.

We will discuss the **Information Flow Log** in detail in Section 5. For now, it suffices to say that it is a new output generated for the cache state machine that counts state transitions that violate the *non-interference* property. For example, external interference occurs when an attacker's cache line is replaced with a victim's cache line.

4.2 Secure Cache Modeling

The state machine of each secure cache architecture described in Section 2.2 can be built based on Figure 1 and Table 2. We now show the differences in the cache models based on the caches' unique features. Details can be found in our technical report [24].

Conventional Cache: Caches start out empty, which means all cache lines are in the **INV** state at the beginning of each experiment, transitioning to either **V** or **A** state on a *V_miss* or *A_miss* event, respectively. Subsequently, they transition between these 2 states. Conventional caches do not invalidate cache lines on load or store instructions. Hence, the *Invalidate* event only occurs on a special "Invalidate addresses" instruction – if this exists in the Instruction Set.

Table 2: Cache structure and state transition

Current State		**Event**	**Information Flow Log**	**Next State**	
State Matrix	*Replacement Matrix*	*Event Vector*	*Interference Matrix*	*State Matrix*	*Replacement Matrix*
$\begin{bmatrix} S_{0,0} & \cdots & S_{0,m-1} \\ \vdots & \ddots & \vdots \\ S_{n-1,0} & \cdots & S_{n-1,m-1} \end{bmatrix}$	$\begin{bmatrix} l_{0,0} & \cdots & l_{0,m-1} \\ \vdots & \ddots & \vdots \\ l_{n-1,0} & \cdots & l_{n-1,m-1} \end{bmatrix}$	$\begin{bmatrix} T_0 \\ \vdots \\ T_{n-1} \end{bmatrix}$	$\begin{bmatrix} I_0 \to O_0 & \cdots & I_{n-1} \to O_0 & I_{-1} \to O_0 \\ \vdots & \ddots & \vdots & \vdots \\ I_0 \to O_{n-1} & \cdots & I_{n-1} \to O_{n-1} & I_{-1} \to O_{n-1} \end{bmatrix}$	$\begin{bmatrix} S'_{0,0} & \cdots & S'_{0,m-1} \\ \vdots & \ddots & \vdots \\ S'_{n-1,0} & \cdots & S'_{n-1,m-1} \end{bmatrix}$	$\begin{bmatrix} l'_{0,0} & \cdots & l'_{0,m-1} \\ \vdots & \ddots & \vdots \\ l'_{n-1,0} & \cdots & l'_{n-1,m-1} \end{bmatrix}$

Static-Partitioning (SP) Cache: For SP cache, the difference with the conventional cache is that each cache line can only have two states (**INV** and **V**, or **INV** and **A**). So transitions of **A** $\rightarrow$ **V** or **V** $\rightarrow$ **A** can never happen.

Partition-Locked (PL) Cache: We consider two uses of PL cache: (1)*PL cache without preload* of the security-critical data before the victim's program begins: the cache is initially empty. Both the victim and attacker can fill the cache with its data. However, once the victim's critical cache lines are locked in the cache, they can not be replaced by the attacker. So we have the transition of **A** $\rightarrow$ **V**, but **V** $\rightarrow$ **A** is forbidden. When the cache is in the state of **V** and encounters the event of *A_miss*, the attacker's data will be sent to the CPU directly, and the cache stays in **V**. (2)*PL cache with preload* of the security-critical data: the victim initially occupies the cache line and locks it in the cache. So neither transition of **A** $\rightarrow$ **V**, nor **V** $\rightarrow$ **A** can happen. The *A_miss* cannot change the state of the PL cache and *A_hit* can never happen for these security-critical lines.

Random-Eviction (RE) Cache: Compared with a conventional cache, the RE cache state machine has two more transitions due to the introduction of random noise: **A** $\rightarrow$ **INV** (attacker's line is randomly chosen to be evicted) and **V** $\rightarrow$ **INV** (victim's line is randomly chosen to be evicted).

Random-Permutation (RP) Cache: An event for RP cache may involve mutiple cache lines when swapping the cache sets. When Line l in set s is in state **V** and encounters an *A_miss*, it still stays in state **V**. Instead, a random Line l' in a random set s' is selected and replaced by the incoming attacker's line, thus jumping to state **A** from whatever state it was in before. All lines of set s and s' in state **A** will be evicted out of the cache and go to state **INV**. 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 l in state **A** encounters a *V_miss*. Line l'' in set s'' is randomly selected and replaced, and all the lines of set s and s'' in state **V** go to **INV** state when swapping sets s and s'' in the victim's *permutation table*.

NewCache: We consider different cache events on a cache line l. When there is a cache hit for the victim (*V_index_hit* & *V_tag_hit*) or the attacker (*A_index_hit* & *A_tag_hit*), the cache will stay in state **V** or **A**, respectively. When there is an index hit but a tag miss for the victim (*V_index_hit* & *V_tag_miss*) or the attacker (*A_index_hit* & *A_tag_miss*) for line l, according to NewCache's replacement policy [14, 21], line l will be directly replaced by the incoming line, jumping from state **V** to **V** or from state **A** to **A**. When there is an index miss for the victim (*V_index_miss*) or the attacker (*A_index_miss*), a random cache line l' is selected to be replaced, and line l' will jump to state **V** or **A** respectively, from whatever state it was in previously.

5. LEAKAGE MEASUREMENT

To evaluate cache systems' side-channel vulnerabilities, we now integrate the leakage model (Section 3) into the cache state machine model (Section 4). We define the *Interference Probability* to measure the information leakage through cache side channels. As an example, we use our method to evaluate Type I attacks (observing cache misses due to external interference) in Table 1.

5.1 Side-channel Leakage Interpretation

As we stated in Section 3, the cache is shared by the victim and the attacker, and is treated as a potential side channel. The victim's *actions* are the inputs to the side channel, and the attacker's *observations* are the outputs from the side channel. To evaluate the side-channel leakage, we need to study the interference between the victim's *actions* and the attacker's *observations*. That is, how the victim's *actions* can affect the attacker's *observations*.

We define I_p as the victim's actions on cache set p ($0 \leq p < n$), and O_q as the attacker's observations of cache set q ($0 \leq q < n$). During the transitions between different cache states, we define a novel **Information Flow Log** to record the side channel's inputs and outputs, as shown in Table 2.

Information Flow Log: This log is used to track the root causes of interference. It defines a structure called **Interference Matrix**. Inside this matrix, $I_p \rightarrow O_q = \{1, 0\}$ depicts if this cache state transition happens with the victim's action on cache set p, which will lead to the attacker's later observation on cache set q. An extra input for noise, I_{-1}, and transition $I_{-1} \rightarrow O_q = \{1, 0\}$ depicts if the inherent noise from the cache channel will lead to the attacker's later observation on cache set q.

The mutual information (Equation 1) and 3 *non-interference* conditions (Equations 2, 3 and 4) can be used to evaluate the side-channel leakage. To do so, we need to go over all the possible cache states and count the number of each kind of interference, $N(I_p \rightarrow O_q)$, where $-1 \leq p < n$ and $0 \leq q < n$. Then we calculate the *Interference Probabilities*:

$$P_{\mathcal{I},\mathcal{O}}(I_p, O_q) = \frac{N(I_p \rightarrow O_q)}{\sum_{-1 \leq p' < n} \sum_{0 \leq q' < n} N(I_{p'} \rightarrow O_{q'})} \quad (5)$$

From Equations 1 and 5 we can evaluate the mutual information between the victim's *actions* and attacker's *observations*. In Equation 6, we only use the range ($0 \leq p' < n$), and omit the noise represented by $p' = -1$, which we consider *fake interference*. The mutual information can accurately reflect the interference between the attacker and the victim through cache side channels.

$$I(\mathcal{I}; \mathcal{O}) = \sum_{0 \leq p' < n} \sum_{0 \leq q' < n} P_{\mathcal{I},\mathcal{O}}(I_{p'}, O_{q'}) \, log \left(\frac{P_{\mathcal{I},\mathcal{O}}(I_{p'}, O_{q'})}{P_{\mathcal{I}}(I_{p'}) P_{\mathcal{O}}(O_{q'})} \right) \quad (6)$$

We can also use the *Interference Probabilities* to identify cases of no information leakage, based on the three *non-interference* conditions in Equations 2, 3 and 4.

C1 (Output Elimination) For all $0 \leq q < n$, $P_{\mathcal{O}}(O_q)$ is close to 0. Then the attacker can not observe any information through the cache side channel, and there is no information leakage.

C2 (Noise Domination) For all $0 \leq q < n$, $P_{\mathcal{I}|\mathcal{O}(I_{-1}|O_q)}$ is close to 1. Then all of the attacker's observations are caused by the channel's noise, the victim's actions do not affect the attacker's observations. Hence there is no information leakage.

C3 (Input Ambiguity) For all $0 \leq p < n$ and $0 \leq q < n$, $P_{\mathcal{I}|\mathcal{O}}(I_p|O_q)$ is close to $P_{\mathcal{I}}(I_p)$. Then the victim's actions cannot be distinguished by the attacker's observations, and there is no information leakage.

5.2 Case Study: Type I Attacks

In Section 2.1 we classify the side-channel attacks into four categories based on the root causes of the attacks: the interference due to cache behaviors. Our cache modeling technique targets the root causes, so it can cover all the categories. As a case study, we pick type I attacks based on cache misses due to external interference, and study different caches' vulnerability to this type of side-channel attacks. Other attack types can be evaluated in a similar way.

For Type I cache attacks, the attacker observes cache misses due to external interference. Whenever a cache line occupied by an attacker is over-written (replaced) by a cache line belonging to a victim, the attacker will have an observation by detecting a cache miss when he next accesses this cache line that he had previously filled with his own data. Specifically, when the victim's cache line with a cache index of p wants to replace the attacker's cache line with a cache index of q (for conventional caches, $p = q$), a cache state transition from **A** to **V** happens, and this will be an *External Interference* (shown in Figure 1). The cause of this *External Interference* is the victim's input I_p to the cache side channel, and the result of this *External Interference* is the attacker's output O_q from the cache side channel later. So this is the interference $I_p \rightarrow O_q$.

It is also possible that the eviction of the attacker's cache line is not due to the victim's actions. This happens in RE cache when the attacker's cache line is randomly selected to be evicted out, or in RP cache when the attacker's cache line is invalidated due to the update of permutation tables. Specifically, when the attacker's cache line with a cache index of q is evicted out of the cache due to such noise, a cache state transition from **A** to **INV** happens, and we call this *Fake Interference* (Figure 1). The cause of this *Fake Interference* is the side channel's inherent noise, and the result of this *Fake Interference* is the attacker's later observations of output O_q from the cache side channel. The interference is labelled $I_{-1} \rightarrow O_q$.

To evaluate the caches' vulnerability to Type I attacks, we count the number of each kind of *External Interference* and *Fake Interference*. Then we use Equation 5 to calculate *Interference Probabilities*. Finally we calculate the mutual information in Equation 6 and use the three non-interference conditions to confirm the absence of information leakage.

We use Murphi [25] to implement our cache security models with the interference property. Murphi is a finite state machine model checker, used to verify the invariants of the system by enumerating all the explicit states. Instead of checking invariants, we use Murphi to go over all the possible cache states and record the **Information Flow Log** for each transition. Without loss of generality, we assume a 3-set, 2-way set-associative conventional cache as the baseline for this study. We used 10 rounds of memory accesses, which was enough to get stable interference probabilities. Murphi will analyze all the possible states, for all the cache lines in the cache.

5.3 Evaluation Results

5.3.1 Analysis of Non-interference Conditions

For each cache, we collect counts for the **Information Flow Log**, including counts of the number of each type of interference. From this, we calculate the joint *Interference Probability* between each input and each output.

Conventional Cache: Table 3 gives the results for the baseline conventional cache. We observe that for a conventional cache, $P_{\mathcal{I},\mathcal{O}}(I_p, O_q)$ is very distinguishable for $(q = p)$ compared to $(q \neq p)$, and none of the three conditions **C1**, **C2** or **C3** are satisfied. So the input $\mathcal{I}$ interferes highly with the output $\mathcal{O}$. We conclude that *the conventional cache is very leaky, and hence insecure.*

Table 3: *Interference Probability* for conventional cache

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_o	I_1	I_2	I_{-1}
O_0	33.3%	0.0%	0.0%	0.0%
O_1	0.0%	33.3%	0.0%	0.0%
O_2	0.0%	0.0%	33.3%	0.0%

(27,996 interferences in total)

Static-Partitioning Cache: Table 4 shows the *Interference Probability* of an SP cache. There are no attacker's observations from SP cache, satisfying condition **C1**. So *SP cache can effectively reduce Type I side-channel leakage to zero.*

Table 4: *Interference Probability* for SP cache

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	0.0%	0.0%	0.0%	0.0%
O_1	0.0%	0.0%	0.0%	0.0%
O_2	0.0%	0.0%	0.0%	0.0%

(0 interferences in total)

Partition-Locked Cache: Tables 5a and 5b display the *Interference Probability* of PL cache without and with preload. PL cache without preload has the same interference distribution as conventional caches, indicating that *PL cache without preload 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 SP cache, indicating that *with proper usage like preloading the victim's sensitive cache lines, PL cache prevents information leakage.*

This example demonstrates the power of our methodology: even though the PL cache without preload may survive the "Prime and Probe" attack during experimentation, our security modeling of PL cache still reveals its vulnerability: there can still be information leakage targeting the cache warm-up stage. This agrees with the observation in [26].

Table 5: *Interference Probability* for PL cache

(a) without preload

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	33.3%	0.0%	0.0%	0.0%
O_1	0.0%	33.3%	0.0%	0.0%
O_2	0.0%	0.0%	33.3%	0.0%

(13,794 interferences in total)

(b) with preload

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	0.0%	0.0%	0.0%	0.0%
O_1	0.0%	0.0%	0.0%	0.0%
O_2	0.0%	0.0%	0.0%	0.0%

(0 interferences in total)

Random-Eviction Cache: Now let us consider the randomization approach. The *Interference Probability* of RE cache is shown in Table 6. The results show that a large amount of interference is fake (70.8%). Although it is still possible for the attacker to retrieve side-channel information from the rest of the interferences (29.2%), it will be a hard job to filter out the noise due to fake interference from the observations. According to **C2**, the larger the proportion of fake interference, the more difficult it is for the attacker to retrieve useful information.

Table 6: *Interference Probability* for RE cache

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	9.7%	0.0%	0.0%	23.6%
O_1	0.0%	9.7%	0.0%	23.6%
O_2	0.0%	0.0%	9.7%	23.6%

(117,349,797 interferences in total)

Random-Permutation Cache: Table 7 displays the Interference Probability for RP cache. We can see that *Fake Interference* constitutes a very high percentage of the total interferences (80.4 %). This is due to the cache line invalidations done when swapping mappings in the *permutation table*. In addition, the interference of each I_p on each O_q has about the same probability. This is due to the random mapping from memory address to cache set. When the attacker observes a cache miss in set q, it is hard for him to tell which set is accessed by the victim. Both *non-interference* conditions **C2** and **C3** hold, thus enhancing *the security of RP cache against Type I side-channel leakage.*

Table 7: *Interference Probability* for RP cache

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	2.17%	2.19%	2.19%	26.8%
O_1	2.19%	2.17%	2.19%	26.8%
O_2	2.19%	2.19%	2.17%	26.8%

(7,842,324 interferences in total)

NewCache: We simulate a NewCache with 3 logical cache lines and 2 physical cache lines. Table 8 shows the Interference Probability. Due to the fully-associative mappings from the Logical Direct Mapped cache to the physical cache, the interference of I_p on any O_q happens with the same probability. According to *non-interference* condition **C3**, the attacker's observations are not affected by the victim's actions. *This means NewCache does not leak information through Type I side-channel attacks.*

Table 8: *Interference Probability* for NewCache

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	11.1%	11.1%	11.1%	0.0%
O_1	11.1%	11.1%	11.1%	0.0%
O_2	11.1%	11.1%	11.1%	0.0%

(1,368,954 interferences in total)

5.3.2 Mutual Information

For each cache architecture, we calculate the mutual information between the victim's actions and the attacker's observations based on Equation 6 (the logarithm base is 2 and information leakage is measured in bits), as shown in Table 9. This shows that the conventional cache and *PL cache without preload* leak the most information. RE cache has a smaller mutual information value, but still gives the attacker some chances to retrieve critical information. The mutual information of RP cache is close to zero, indicating that it will be hard for the attacker to leak secrets with Type I side-channel attacks. SP cache, *PL cache with preload* and NewCache have zero mutual information, so the attacker can not get the victim's secrets from his observations.

The conclusions from mutual information are consistent with our previous analysis of non-interference *conditions.*

Table 9: Mutual Information for each cache architecture

Cache Architecture	$I(I,O)$ (bits)
Conventional	1.585
SP	0.000
PL-w/o preload	1.585
PL-w/ preload	0.000
RE	0.461
RP	2.586×10^{-6}
New	0.000

5.4 Discussion

From the above analysis we observe that different defenses usually focus on different *non-interference* conditions. For the partitioning approach, the defenses usually try to realize **C1**. The attacker cannot observe output from the channel as it is isolated from the victim (SP cache and PL cache). For the randomization approach, the defenses usually try to realize **C2** and **C3**. For **C2**, the cache adds a large amount of random noise to the attacker's observations (RE cache and RP cache). For **C3**, the cache randomizes the mappings between the victim's actions and the attacker's observations (RP cache and NewCache). This ambiguity makes it hard for the attacker to get accurate conclusions about the victim's actions. Any of the three conditions is effective at reducing the side-channel leakage. We hope this will inspire researchers to propose more defenses beyond the partitioning and randomization approaches discussed in this paper.

Our modeling methodology provides different usages: (1) a general evaluation of a cache's vulnerability to side-channel attacks, as in this paper, considers all possible cache state transitions for successive rounds of memory accesses. This will cover all possible attacks on all ciphers; (2) an evaluation of a cache's vulnerability to a specific attack on a specific cipher, can be achieved by feeding the cache models with the victim's and attacker's actual memory access traces.

6. MODELING OTHER ATTACKS

We discuss how to apply our modeling methodology to other cache side-channel attacks and cache features.

6.1 Other Cache Attacks

Our case study focused on Type I attacks: the cache misses that cause external interference between the attacker and the victim. So we consider the transitions of *V_miss*, from state **A** to **V**. Our model can also be applied to the other three attack categories in Table 1.

(1) *Type II Attacks*: These are based on cache misses due to internal interference, so we consider the transitions of *V_miss* from state **V** to **V**. The channel's input I_p is the victim's access with a cache index of p, and the output O_q is the victim's replaced cache line with a cache index of q. A random cache mapping for the victim can make $I_p \rightarrow O_q$ ambiguous, reducing the vulnerability to leak information.

(2) *Type III Attacks*: These are based on cache hits due to external interference. We need a fourth type of state **A/V**, indicating this line contains a memory line shared by the victim and the attacker. Then we consider the transitions of *A_hit* from state **A/V** to **A/V**. The channel's input I_p is the victim's access of a shared line with address index p, and the output O_q is the event that the attacker gets a cache hit in the cache set q. $I_p \rightarrow O_q$ denotes the interference that the attacker's line with address index p gets a cache hit in the cache set q, due to the victim's placement of this shared line. We denote I_{-1} as the event that brings the victim's line into

the cache for non-critical operations like prefetching, then $I_{-1} \to O_q$ is a *Fake Interference* which can introduce noise.

(3) *Type IV Attacks*: These are based on cache hits due to internal interference. We consider the transitions of *V_hit* from state **V** to **V**. The input I_p to the channel is the event that the victim brings its line with index p into the cache, and the output O_q is the event that the victim gets a cache hit when accessing a cache set q. $I_p \to O_q$ denotes the interference that the victim's line with address index p gets a cache hit in the cache set q. Similarly I_{-1} is the non-critical operations that bring in the victim's cache line, then $I_{-1} \to O_q$ is a *Fake Interference* that adds random noise.

Relevant interference probabilities, mutual information and non-interference conditions can similarly be evaluated for these types of side-channel vulnerabilities.

6.2 Other System Features

Our cache modeling methodology can also be extended to model other features of modern microprocessors.

(1) *Multiple processes*: Other processes, not the victim nor the attacker, can affect the attacker's observations. We can introduce a new state **R** to indicate the cache line occupied by the Rest of the processes. A new cache event "*R_miss*" can transition the cache state from **A** to **R**, generate *Fake Interference* and add noise to the attacker's observations. **R** can also be superimposed on the **INV** state, and "*R_miss*" on the "*Invalidate*" transition from **A** to **INV** states, in Figure 1. This generalizes the *Fake Interference* we measure as "noise" coming from multiple sources - invalidations, other processes or other channel noise.

(2) *Prefetching*: This fetches data into cache before being requested, to avoid future cache misses. We add two new events "*V_prefetch*" and "*A_prefetch*" to model the data prefetching of victim and attacker. Prefetching can reduce the side-channel leakage by adding noise. For instance, in a Type I attack, *V_prefetch* can evict the attacker's cache line and transition the cache state from **A** to **V**. This generates *Fake Interference* when the attacker has a cache miss later for this line, and may satisfy Condition **C2**. *A_prefetch* can evict the victim's cache line and transition the cache state from **V** to **A**. Then the attacker will experience a cache hit for this line, and cannot observe the victim's previous actions. This may satisfy Condition **C1**.

(3) *AES-NI* [27]: Intel x86 processors introduced new instructions specifically for AES block encryption and decryption. This eliminates cache usage for those AES implementations rewritten using AES-NI instructions. Since this removes state **V** and all transitions related to it from Figure 1, the attacker cannot observe any interference from the victim and Condition **C1** is satisfied. Unfortunately, AES-NI does not apply to legacy code (or new code written without using AES-NI instructions). It does not mitigate the cache side-channel leakage from other ciphers, e.g., RSA. It also does not protect cache side-channel leakage in other processors, e.g., the dominant ARM processors used in mobile devices.

7. VALIDATION OF CACHE MODELS

We launch an actual attack program to see if this validates the results of our cache modeling case study. We use the *Probability Distribution of Candidate Keys* to quantify the feasibility of this attack, hence the system's vulnerability.

7.1 Probability Distribution of Candidate Keys

When an attacker attempts to break a cryptography computing platform, he can feed different plaintexts into the platform, and ask for encryption with the key he wants to steal. He then collects the side-channel observations during the encryptions and tries to infer the keys or narrow down its possible values. To improve the accuracy and fully recover the keys, the attacker usually repeats a bunch of attack rounds to obtain different candidate keys. He calculates the *Probability Distribution of Candidate Keys*, and selects the key with the highest probability. A successful attack is able to select the correct key based on its significantly higher probability in the *Probability Distribution of Candidate Keys*. So we use this to evaluate the success of an attack, and hence the systems' vulnerability to this attack.

7.2 Implementation

We launch an access-based side-channel attack on AES [17]. We use gem5 [28] to simulate this attack on different cache architectures, and compare their *Probability Distribution of Candidate Keys*. For each cache, we simulate L1 caches, with cache size of 32 Kbytes, line size of 32 bytes and set-associativity of 8-way, which is a typical configuration in modern processors. The victim runs AES encryption for 2^{18} random blocks in the *study* phase (the key is known to the attacker) and in the *attack* phase (the key is unknown to the attacker). The attacker *primes and probes* the cache continuously to collect the access time for each cache line, to infer the victim's memory accesses, and hence his encryption key.

7.3 Attack Results

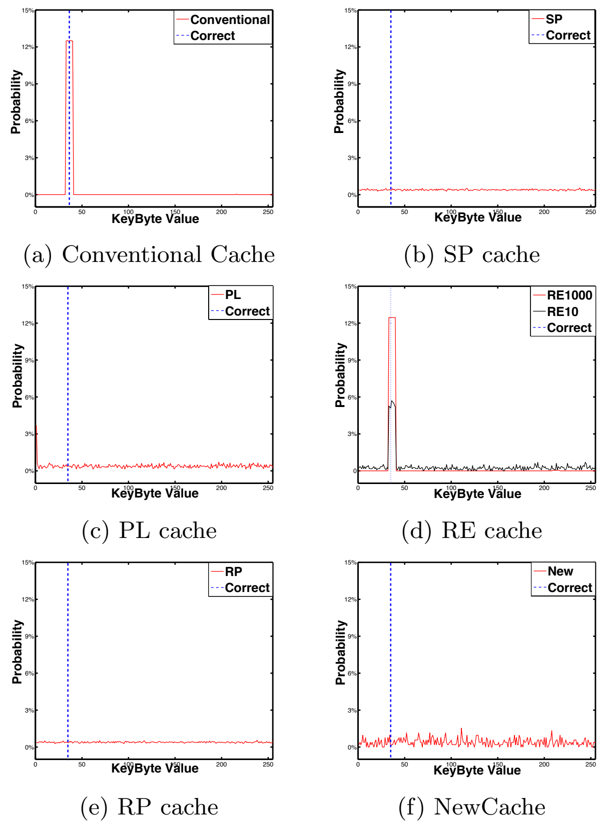
Figure 2 shows the attack results. We use a solid red line to show the *Probability Distribution of Candidate Keys*, and a dotted blue line to denote the correct encryption key.

For conventional caches (Figure 2a), eight keys (32-39) get more than 10 % probabilities 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 because one cache line contains 8 AES entries, and the attacker is unable to differentiate which entry is actually accessed, when he observes a victim's access to this line. He needs other methods like brute-force or two-rounds attack [17] to find the correct key from the 8 possible values. We conclude this attack on the conventional cache succeeds.

Similar results can be observed for Random Eviction cache (Figure 2d). This shows two types of RE cache: RE1000 (a random cache line is evicted every 1000 memory accesses – in red) and RE10 (a line is evicted every 10 memory accesses – in black). RE cache also leaks eight candidate keys: RE1000 has the same probability distribution as conventional caches, while RE10 is much smaller. A RE cache with more frequent random evictions is more difficult to attack.

Figure 2b shows the distribution of candidate keys for SP cache. This shows that the attack does not produce any distinguished candidate keys. Thus this attack on SP cache fails at 2^{18} samples. We get the same conclusion for Partition-Locked Cache (Figure 2c), Random-Permutation Cache (Figure 2e) and Newcache (Figure 2f).

Comparing Table 9 and Figure 2, we see that the results we get from the two independent evaluation methods are consistent. For the partitioning approach, PL and SP caches can effectively defend against Type I side-channel attacks. For the randomization approach, RP and NewCache are also very effective in reducing Type I side-channel leakage. RE



(a) Conventional Cache (b) SP cache (c) PL cache (d) RE cache (e) RP cache (f) NewCache

Figure 2: Probability Distribution of Candidate Keys

caches are attackable, but if the eviction frequency increases (at the cost of performance), the attack becomes harder.

In several cases, our abstract cache models give more information than the actual attacks. For example, they can show the effect of PL cache with preloading of the sensitive table data (not vulnerable) versus PL cache with no preloading (vulnerable), as shown in Table 5.

8. RELATED WORK

Past work in evaluating side-channel attacks can be classified into several categories based on their features:

Mutual Information: [29] applied mutual information and Guessing Entropy to evaluate the feasibility of key-recovery. [30] proposed static analysis to establish formal security guarantees against cache side-channel attacks, and can estimate the upper bound of information leakage from side channels. [31] presented an information-theoretic metric for adaptive side-channel attacks, which can estimate the attacker's remaining uncertainty for adjusting his strategy.

Success Probability: In [29], Success Rate is defined as a general metric to evaluate the feasibility of side-channel key recovery. Then [32] 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. [33] builds a predictive model for evaluating the side-channel leakage through caches. It calculates the probability that the attacker can correctly detect a memory access given a victim's critical memory access.

Correlation Metric: [34] proposed the Side-channel Vulnerability Factor (SVF) to measure a system's vulnerability to all side channels. It calculates the Pearson Correlation Coefficient between the Similarity Matrices of the victim's execution traces and the attacker's observation traces. [35] proposed the timing-SVF metric for timing-based cache side-channel attacks, which [34] did not address. In [36], a metric called Cache Side-channel Vulnerability (CSV) is designed to overcome SVF's issues in its scope, definition and measurements. It also states that using a single metric like SVF [34] to evaluate the system's vulnerability to all possible forms of side-channel information leakage is problematic as it may give misleading results and furthermore, it does not correctly determine which secure cache designs are more effective in defending against which side-channel attacks.

Formal Verification: Porras and Kemmerer designed the technique of covert flow trees to systematically detect and identify covert channels between processes [37]. [38] built models of timing side-channel leakage from the program code level. [39] uses the technique of self-composition to verify the non-interference properties of cryptographic software by considering two copies of the program.

Most of the above methods aim to evaluate the feasibility of the attacker's behaviors, instead of the system's intrinsic vulnerability. Unlike these past methods, we are the first to model the cache architectures and measure their leakage.

9. CONCLUSIONS

This work proposes a novel methodology to evaluate a cache system's vulnerability to side-channel attacks. We model side channel leakage from the *non-interference* property, and use mutual information with three *non-interference* conditions to guarantee no side-channel leakage. We then show how to model cache architectures, and integrate these with our side-channel leakage model. We also perform a real attack on each of our detailed secure cache architecture simulations, to see if our model is consistent with reality ("ground truth"), to validate our modeling methodology.

Our modeling methodology focuses on the root cause of cache side-channel leakage: the interference impacting cache behavior. It can theoretically cover all types of side-channel attacks (known or unknown). In the case study, we consider the side-channel attacks (Type I) based on cache misses due to external interference. But these models of caches and interferences can be extended to other types of attacks, as discussed in Section 6. Future work can also extend these evaluation methods to new cache architectures and other system features impacting cache behavior, as well as to other subsystems (not just caches) that may be vulnerable to side-channel or covert channel attacks.

10. ACKNOWLEDGMENTS

This work was supported in part by DHS/AFRL FA8750-12-2-0295 and NSF CNS-1218817.

11. REFERENCES

[1] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *IEEE Symp. on Security and Privacy*, 2011.

[2] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Investigations of power analysis attacks on smartcards," in *USENIX Workshop on Smartcard Technology*, 1999.

[3] P. C. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *Annual Intl. Cryptology Conference on Advances in Cryptology*, 1999.

[4] N. Homma, T. Aoki, and A. Satoh, “Electromagnetic information leakage for side-channel analysis of cryptographic modules,” in *IEEE Intl. Symp. on Electromagnetic Compatibility*, 2010.

[5] P. Kocher, R. Lee, G. McGraw, and A. Raghunathan, “Security as a new dimension in embedded system design,” in *Design Automation Conference*, 2004.

[6] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, “A practical implementation of the timing attack,” in *Intl. Conf. on Smart Card Research and Applications*, 2000.

[7] C. Percival, “Cache missing for fun and profit,” in *Proc. of BSDCan*, 2005.

[8] D. J. Bernstein, “Cache-timing attacks on aes,” tech. rep., 2005.

[9] J. Bonneau and I. Mironov, “Cache-collision timing attacks against aes,” in *Lecture Notes in Computer Science series 4249*, Springer, 2006.

[10] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-vm side channels and their use to extract private keys,” in *ACM Conference on Computer and Communications Security*, 2012.

[11] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, “Software mitigations to hedge aes against cache-based software side channel vulnerabilities,” 2006.

[12] E. Käsper and P. Schwabe, “Faster and timing-attack resistant aes-gcm,” in *Cryptographic Hardware and Embedded Systems*, 2009.

[13] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” in *ACM/IEEE Intl. Symp. on Computer Architecture*, 2007.

[14] Z. Wang and R. Lee, “A novel cache architecture with enhanced performance and security,” in *IEEE/ACM Intl. Symp. on Microarchitecture*, 2008.

[15] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, “Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks,” *ACM Trans. Archit. Code Optim.*, 2012.

[16] O. Aciiçmez and c. K. Koç, “Trace-driven cache attacks on aes,” in *Intl. Conference on Information and Communications Security*, 2006.

[17] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of aes,” in *RSA conference on Topics in Cryptology*, 2006.

[18] O. Aciiçmez, “Yet another microarchitectural attack: exploiting i-cache,” in *ACM workshop on Computer security architecture*, 2007.

[19] Z. Wang, *Information Leakage Due to Cache and Processor Architectures.* PhD thesis, Princeton, 2012.

[20] Z. Wang and R. B. Lee, “Covert and side channels due to processor architecture,” in *Annual Computer Security Applications Conference*, 2006.

[21] F. Liu and R. B. Lee, “Security testing of a secure cache design,” in *Hardware and Architectural Support for Security and Privacy*, 2013.

[22] J. A. Goguen and J. Meseguer, “Security policies and security models,” in *IEEE Symp. on Security and Privacy*, 1982.

[23] T. M. Cover and J. A. Thomas, *Elements of Information Theory.* Wiley InterScience, 2006.

[24] T. Zhang and R. B. Lee, “Secure Cache Modeling for Measuring Side-channel Leakage,” in *Tech. Report, http://palms.ee.princeton.edu/node/428.*

[25] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, “Protocol verification as a hardware design aid,” in *Intl. Conference on Computer Design: VLSI in Computer & Processors*, 1992.

[26] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, “Hardware-software integrated approaches to defend against software cache-based side channel attacks,” in *IEEE Intl. Symp. on High Performance Computer Architecture*, 2009.

[27] S. Gueron, “Intel advanced encryption standard (aes) instructions set,” 2010.

[28] “The gem5 simulator system,” in *http://www.gem5.org*.

[29] F.-X. Standaert, T. G. Malkin, and M. Yung, “A unified framework for the analysis of side-channel key recovery attacks,” in *Annual Intl. Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques*, 2009.

[30] B. Köpf, L. Mauborgne, and M. Ochoa, “Automatic quantification of cache side-channels,” in *Intl. Conference on Computer Aided Verification*, 2012.

[31] B. Köpf and D. Basin, “An information-theoretic model for adaptive side-channel attacks,” in *ACM Conf. on Computer and Comms. Security*, 2007.

[32] C. Rebeiro and D. Mukhopadhyay, “Boosting profiled cache timing attacks with a priori analysis,” *IEEE Trans. on Information Forensics and Security*, 2012.

[33] L. Domnitser, N. Abu-Ghazaleh, and D. Ponomarev, “A predictive model for cache-based side channels in multicore and multithreaded microprocessors,” in *Intl. Conference on Mathematical Methods, Models and Architectures for Computer Network Security*, 2010.

[34] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, “Side-channel vulnerability factor: a metric for measuring information leakage,” in *ACM/IEEE Intl. Symp. on Computer Architecture*, 2012.

[35] S. Bhattacharya, C. Rebeiro, and D. Mukhopadhyay, “Hardware prefetchers leak: A revisit of SVF for cache-timing attacks,” in *Hardware and Architectural Support for Security and Privacy*, 2012.

[36] T. Zhang, S. Chen, F. Liu, and R. B. Lee, “Side channel vulnerability metrics: the promise and the pitfalls,” in *Hardware and Architectural Support for Security and Privacy*, 2013.

[37] P. Porras and R. Kemmerer, “Covert flow trees: a technique for identifying and analyzing covert storage channels,” in *IEEE Computer Society Symp. on Research in Security and Privacy*, 1991.

[38] J. Svenningsson and D. Sands, “Specification and verification of side channel declassification,” in *Intl. Conf. on Formal Aspects in Security and Trust*, 2010.

[39] J. Bacelar Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, “Formal verification of side-channel countermeasures using self-composition,” *Sci. Comput. Program.*, 2013.