Analyzing Cache Side Channels Using Deep Neural Networks

Tianwei Zhang
No Affiliation
tianweiz@alumni.princeton.edu

Yinqian Zhang
The Ohio State University
yinqian@cse.ohio-state.edu

Ruby B. Lee
Princeton University
rblee@princeton.edu

ABSTRACT

Cache side-channel attacks aim to breach the confidentiality of a computer system and extract sensitive secrets through CPU caches. In the past years, different types of side-channel attacks targeting a variety of cache architectures have been demonstrated. Meanwhile, different defense methods and systems have also been designed to mitigate these attacks. However, quantitatively evaluating the effectiveness of these attacks and defenses has been challenging. We propose a generic approach to evaluating cache side-channel attacks and defenses. Specifically, our method builds a deep neural network with its inputs as the adversary's observed information, and its outputs as the victim's execution traces. By training the neural network, the relationship between the inputs and outputs can be automatically discovered. As a result, the prediction accuracy of the neural network can serve as a metric to quantify how much information the adversary can obtain correctly, and how effective a defense solution is in reducing the information leakage under different attack scenarios. Our evaluation suggests that the proposed method can effectively evaluate different attacks and defenses.

ACM Reference Format:
Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 2018. Analyzing Cache Side Channels Using Deep Neural Networks. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274694.3274715

1 INTRODUCTION

Side-channel attacks are serious security threats to the confidentiality of computing. They exploit the physical characteristics of the system to breach the confidentiality of the victim's applications under the cryptography or access control protection. Generally, when a victim program executes critical operations with secret information, the host system may exhibit secret-dependent features or behaviors that can be observed by the side-channel adversary. Typical examples of such features include power consumption [24], electromagnetic radiation [39], acoustic noise [15], timing [37] *etc.* Based on these features, the adversary is able to infer the secrets.

Among all the potential sources of information leakage, cache side channels are particularly dangerous. The interferences between different programs on the cache usage can be exploited by the adversary to steal information [4, 6, 19, 37]. Past work have shown the possibility and practicality of cache side-channel attacks in cloud servers [57, 58], mobile devices [29, 56] and web browsers [35]. Recently, two critical processor vulnerabilities, Meltdown [30] and Spectre [23], were discovered which allow user-level programs to break the OS protection ring and steal data from the privileged kernel space or other programs. These two vulnerabilities exploit cache side channels or covert channels to transmit information.

A variety of cache attacks have been proposed targeting different victim applications (*e.g.*, cryptographic programs [36, 37], the user Interface [18], web applications [35, 58]), system configurations (*e.g.*, non-virtualization [19, 37], virtualization [34, 57]) and CPU cache architectures (*e.g.*, L1 cache [37, 57], LLC [21, 34, 52]). Meanwhile, different defenses haven been designed to mitigate these attacks by modifying system software [22, 44, 59, 60] or hardware [13, 31, 32, 49, 51]. In order to evaluate these attacks or defenses, the most straightforward way is to implement the real attacks or defense solutions, and check if the adversary can steal the secrets. While this method can provide reliable and convincing results, it is not easy to conduct experimental evaluation. First, reproducing side-channel attacks from literature usually takes a lot of effort, as these attacks are delicate and complex. It is also difficult to implement the defenses, especially for the hardware solutions. Second, given the fact that there are so many types of attacks and defenses, it is extremely difficult to experimentally evaluate different attacks against different solutions to get comprehensive conclusions.

To efficiently analyze cache side-channel attacks and defenses, different metrics [10, 25, 40, 45, 54] and models [20, 55] were proposed. However, they have some limitations. (1) Some work only consider one specific attack and cannot be generalized to all known cache side-channel attacks. (2) Some metrics [10, 54] assume that the adversary's observation and victim's execution are linearly correlated, which reduces the attack scope. (3) Some metrics [25, 40, 45] need to model how the adversary analyzes the side-channel information and recovers the keys, which makes them more complicated and less practical. (4) Some methods (e.g., [55]) suffer from state explosion problem and can only be used to simulate very simple cache activities.

In this paper, we propose a novel method to quantify the effectiveness of cache side-channel attacks and defenses by using deep neural networks. Our key insight is that the essence of cache side-channel attacks is to *learn* the relationship between the adversary's observations and victim's execution traces, and *predict* the sensitive information from the observations. This learning and prediction process can be naturally simulated by a neural network. Past work have shown the possibility of using machine learning methods to help the adversary recover cryptographic keys in some side-channel attacks [2, 7, 19, 35, 57]. Different from those work, we aim to build a generic deep learning based approach to analyze different types of cache side-channel attacks as well as defenses.

Specifically, the adversary's side-channel observations are modeled as inputs, and the victim's execution traces are modeled as

ACSAC '18, December 3–7, 2018, San Juan, PR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274715

outputs. These data are fed into a deep neural network and the analysis is performed in two phases: in the *training* phase, a neural network model is trained, which discloses the relationship between the inputs and outputs. This training process simulates how the adversary learns the relationship between side-channel observations and sensitive information. In the *inference* phase, the input data is fed into the trained model and the output data is generated, which simulates how the adversary predicts sensitive information from side-channel observations. *The prediction accuracy can serve as a metric to quantify the effectiveness of the attacks and defenses.*

Our method has several advantages. First, the neural network can automatically discover the relationship between the adversary's observation and the victim's execution. So we do not need to consider and model the process of secret recovery for each specific attack method. Second, the neural network can reveal both linear and non-linear relationships. The non-linear relationship significantly improves over previous methods, such as Zhang et al. [54] and Demme et al. [10]. Third, it is feasible to build state machines to model cache activities and generate training and inference datasets for the neural network. So there is no need to run actual attacks in real systems for data collection, which saves a lot of time and effort for cache side-channel analysis. Fourth, our method is generic and can be applied to different attacks and defenses. We show in this paper how we use this method to evaluate five common existing attack techniques (i.e., Prime-Probe, Evict-Time, Flush-Reload, Flush-Flush, Prime-Abort) and two categories of defense strategies (i.e., isolation, randomization).

The key contributions of this paper are:

- A novel and generic method to analyze cache side channels using deep neural networks.
- Modeling of cache side-channel attack techniques and defense strategies using finite state machines.
- Evaluations and comparisons of different attacks and defenses.

The rest of the paper is organized as follows: Sec. 2 gives the background of cache side-channel attacks, and deep neural networks. Sec. 3 gives the methodology overview. Sec. 4 describes how to use a finite-state machine to model cache activities and generate datasets. Sec. 5 describes how we conduct the side-channel analysis using a deep neural network. Sec. 6 presents the evaluation of our method. Sec. 7 validates our methodology using real attacks. We discuss related work in Sec. 8 and conclude in Sec. 9.

2 BACKGROUND

2.1 Cache Side-channel Attacks

CPU caches provide a side channel for information leakage from the victim's program to the adversary's program. The root cause is the interferences (timing or functional) between the victim and adversary on the cache usage. As the victim executes on the system, it may have different cache behaviors when accessing the memory. These behaviors can interfere with the adversary and cache states. Then the adversary tries to observe his own cache state or the victim's external behaviors (*e.g.*, execution time) to deduce the victim's cache behaviors, which might help him steal the secrets.

To capture the victim's memory traces, the adversary usually conducts two stages. We abstract these as Set-Check. In the Set stage the adversary sets some critical cache blocks or sets to certain states. Then the adversary waits for some time for the victim to execute. If the victim accesses any of these cache blocks or sets, it will change the cache states. In the Check stage, the adversary checks the states of the cache blocks or sets to infer the victim's memory traces. He can repeat the Set-Check stages until he obtains enough victim's memory traces to recover the confidential data. Based on the techniques of setting and checking cache states, we can classify the cache side-channel attacks into three categories:

Contention-based attacks. The adversary checks if the victim's memory accesses cause cache contention with his own activities, and uses such contention information to deduce the victim's traces.

Prime-Probe attack [37]: the adversary selects an array of memory blocks that can exactly fill up certain cache sets, which contain the victim's secret. Then in the Prime stage, the adversary reads each memory block in the array to evict all the victim's data in these cache sets. The adversary waits for a certain time and then performs the Probe stage, where he again reads each memory block in the array, and measures the time of each memory access. A large access time means cache contention occurs, indicating that this cache set has been accessed by the victim between the Prime and Probe stages. The Probe stage is the Prime stage for the next round.

Evict-Time attack [36]: similar to the Prime-Probe attack, the adversary also prepares an array of memory blocks that can fill up the critical cache sets. The Evict stage is the same as the Prime stage, except that typically only one cache set is evicted. After this stage the victim executes certain blocks of code (*e.g.*, encryption of one plaintext). The adversary measures the victim's execution time as his second stage Time. A large execution time means that the victim accesses the critical cache sets during the execution, and generates cache contention with the adversary.

Reuse-based attacks. This type of attacks assumes that the system enables the Kernel Samepage Merging (KSM) technique [1], where identical memory pages amongst different processes are merged and shared. So the adversary and victim processes can share the same pages containing cryptographic codes. The adversary checks if some memory blocks inside these shared memory pages are used by the victim to infer the victim's memory access.

Flush-Reload attack [19]: the adversary selects an array of critical memory blocks from these shared pages. In the flush stage, the adversary flushes these memory blocks out of the entire cache hierarchy. Then he waits for a certain interval. In the reload stage, the adversary accesses these memory blocks and measures the latency. A short access time for one memory block indicates a cache hit, as this block has been reused by the victim and brought into the cache during the interval.

Flush-Flush attack [17]: the setup and first stage is the same as Flush-Reload. In the second stage, instead of reloading the shared memory blocks, the adversary still flushes the blocks. If the victim fetches a block into the cache, then flushing this block will take a longer time than when it is out of the cache. So Flush has the same effect as Reload. Besides, a single Flush operation can serve as Check for the current round as well as Set for the next round.

Abort-based attacks. Different from the above two timing-based attacks, the adversary can use the occurrence of aborts to infer the victim's access. Specifically, in an Intel Transactional Memory (TSX) processor, when the victim evicts the adversary's transactional memory block out of the cache, the adversary receives an abort. This serves as an indication of the victim's access to a certain cache set.

PRIME-ABORT attack [11]: the adversary prepares an array of memory blocks that can fill up a target cache set. The first stage is also similar to the PRIME-PROBE attack, except that the adversary also opens a TSX transaction first for his memory blocks. Then he can just wait for the occurrence of the second stage ABORT. When the victim evicts the adversary's block out of the cache, the adversary observes an abort and he can detect the victim's access.

2.2 Cache Side-channel Defenses

Different defense systems and methods have been proposed to protect against cache side-channel leakage. Basically these solutions follow one of two strategies: isolation and randomization [50].

Isolation. The root cause of cache side-channel attacks is due to the interference in the physical cache regions (PRIME-PROBE, EVICT-TIME, PRIME-ABORT) or memory pages (FLUSH-RELOAD, FLUSH-FLUSH). So one straightforward approach is to isolate the adversary's and victim's cache activities. To isolate cache regions, the cache can be partitioned into different zones by sets or ways via hardware [13, 31, 51] or software methods [22, 44], and allocate these zones to different programs. This can effectively prevent information leakage due to cache set interference, at the cost of performance degradation. To isolate memory blocks, we can disable memory page sharing or memory deduplication [52, 58]. This can defeat reuse-based attacks, at the cost of memory space waste.

Randomization. This strategy aims to introduce randomness to the adversary's measurements to make it hard for him to distinguish the victim's cache usage. Typical examples of this strategy include random memory-to-cache mappings [49, 51], randomized cache prefetches [32], timers [28, 48] and cache states [59]. For instance, a random eviction cache [55] periodically selects a random cache block to evict. This adds faked cache activities into the adversary's observation, and he cannot distinguish them from the victim's activities. A random permutation cache [51] or a NewCache [33, 49] dynamically randomizes the memory-to-cache mapping for each process. When the adversary obtains the victim's cache access trace, he cannot recover the victim's memory access trace, as he does not know the victim's memory-to-cache mapping.

2.3 Deep Neural Networks

A deep neural network is a parameterized function $f_\theta : \mathcal{X} \mapsto \mathcal{Y}$ that maps an input tensor $x \in \mathcal{X}$ to an output tensor $y \in \mathcal{Y}$. Various neural network architectures have been proposed and applied to different tasks, *e.g.* multilayer perceptrons [42], convolutional neural networks [27] and recurrent neural networks [43].

We use multilayer perceptrons as an example. A neural network usually consists of an input layer, an output layer and a sequence of hidden layers between the input and output. Each layer is a collection of units called *neurons*, which are connected to other neurons in the previous layer and the following layer. Each connection between the neurons can transmit a signal to another neuron in the next layer. In this way, a neural network transforms the inputs through hidden layers and then the outputs, by applying a linear function (weight: W_i, $bias : b_i$) followed by an element-wise nonlinear activation function ϕ_i (*e.g.* `sigmoid`, `ReLU` or `softmax`) in each layer i, as shown in Equation 1.

$$\begin{aligned} h_1 &= \phi_1(W_1 x + b_1) \\ h_2 &= \phi_2(W_2 h_1 + b_2) \\ &\cdots \\ h_n &= \phi_n(W_n h_{n-1} + b_n) \\ y &= \phi_{n+1}(W_{n+1} h_n + b_{n+1}) \end{aligned} \tag{1}$$

Model training. The training process of a neural network is to find the optimal parameters θ that can accurately reflect the relationship between $\mathcal{X}$ and $\mathcal{Y}$. To achieve this, a training dataset $D^{train} = \{x_i^{train}, y_i^{train}\}_{i=1}^N$ with N samples is needed, where $x_i^{train} \in \mathcal{X}$ is the feature data and $y_i^{train} \in \mathcal{Y}$ is the corresponding ground-truth label. Then a loss function L is adopted to measure the distance between the ground-truth output y_i^{train} and the predicted output $f_\theta(x_i^{train})$. The goal of training a neural network is to minimize this loss function (Equation 2). Backward propagation [16] and stochastic gradient descent [41] are commonly used methods to approximately achieve this goal. The optimal parameters θ^* together with the network topology form the deep learning model.

$$\theta^* = \arg\min_\theta \left(\sum_{i=1}^N L(y_i^{train}, f_\theta(x_i^{train})\right) \tag{2}$$

Model inference. After the model training is completed, given an input x, the corresponding output can be calculated as $y = f_{\theta^*}(x)$. This prediction process is called inference. We can also calculate the prediction accuracy of the model over a testing dataset $D^{test} = \{x_i^{test}, y_i^{test}\}_{i=1}^N$ to measure the model's performance. For a classification problem where the output is a discrete number of values, the prediction accuracy is defined in Equation 3, where $\mathbb{I}$ is the indicator function, i.e., $\mathbb{I}(a = b) = 1$ when $a = b$, and 0 when $a \neq b$.

$$acc(D^{test}, f_{\theta^*}) = \frac{1}{N}\sum_{i=1}^N \mathbb{I}(f_{\theta^*}(x_i^{test}) = y_i^{test}) \tag{3}$$

3 METHODOLOGY OVERVIEW

We aim to provide a generic methodology to quantitatively measure and compare the side-channel adversary's capabilities, as well as the effectiveness of defense solutions. To achieve this, deep neural networks are adopted to conduct the side-channel analysis and evaluation. The key insight that motivates the design of this methodology is that the goal of a deep neural network is to *learn* the relationship between feature variables and label variables given a number of training data, and then use this relationship to *predict* the label from a given feature. This exactly matches the goal of a side-channel adversary: *learning* the relationship between the side-channel observation and sensitive information, and then *predicting* the victim's sensitive information from the adversary's observations. Therefore, *a side-channel attack can be modeled as a deep neural network: the input features of the network are the adversary's observation, and the output labels of the network are the victim's execution information.*

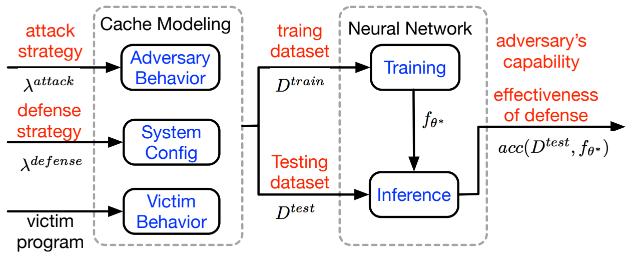


Figure 1: Methodology overview.

The state-of-the-art deep learning methods can discover such relationships automatically, and the prediction accuracy can serve as a metric for evaluating how much information is leaked via the side channel under a specific attack and system setting.

A key technical challenge in our design is how to generate the dataset for neural network training and evaluation. One method is to collect memory traces from real-world attacks in real systems. While feasible, this method requires implementations of side-channel attacks and defenses, which are difficult to obtain (e.g., new hardware designs) and usually very time-consuming. An alternative solution is to collect memory traces from simulating attacks and defenses. However, a cycle-accurate simulator is extremely slow and can take several days for simulating a side-channel attack. To solve this challenge, we propose to use finite-state machines to model the key characteristics of different attacks and defenses, and generate neural network datasets. Specifically, the status change of a cache block (for a reuse-based attack) or a set (for a contention-based attack or an abort-based attack) is modeled as a state machine. The adversary's and victim's activities can change the states of the cache block or set. Then the cache in consideration is modeled as the combination of all state machines of its sets or blocks. The adversary's observation and victim's memory activities are collected during the state machine execution. Using this method we can quickly collect enough data to train and evaluate the neural network.

Figure 1 shows the overview of our methodology. It consists of two steps. The first step is to construct the datasets for the neural network by state machine modeling. We provide a generic method to build state machines for different attacks and defenses. To build a state machine, one attack strategy λ^{attack} (Equation 4) and one defense strategy $\lambda^{defense}$ (Equation 5) need to be provided. The definitions of each term in the two strategies can be found in Table 1. The attack strategy will affect the adversary's behaviors and the defense strategy will affect the system configurations during cache modeling. We discuss the impacts of these factors in Section 4.1, and evaluate the effectiveness of the strategies in Section 6.

The output of the cache modeling step is a training dataset D^{train} and testing dataset D^{test}. If the goal is to measure the adversary's capability in a side-channel attack, the corresponding attack strategy λ^{attack} is specified, while the defense strategy $\lambda^{defense}$ is set as a conventional cache (*i.e.*, set P as null). Then the built state machine and generated datasets can reflect the adversary's capability of stealing information. If the goal is to quantify the effectiveness of a defense solution, the corresponding defense strategy $\lambda^{defense}$ is specified, while the attack strategy λ^{attack} is maximized (*i.e.*, set the highest speed S, coverage C and attack duration D). Then the state machine and datasets can reflect the resilience of the defense system against the strongest attack.

Attack Strategy	T	Attack technique of SET-CHECK (Sec. 2.1)
	S	Relative speed of adversary's memory access to the victim's memory access
	C	Percentage of the critical cache/memory region the adversary can cover
	D	Attack duration the adversary needs to discover the side-channel relationship.
Defense Strategy	A	Cache architecture, including number of cache sets, set-associativity, *etc.*
	P	Cache security policy. This can include cache isolation or randomization (Sec. 2.2)

Table 1: Parameters of attack strategies and defense strategies in modeling side-channel attacks

$$\lambda^{attack} = \{T, S, C, D\} \tag{4}$$

$$\lambda^{defense} = \{A, P\} \tag{5}$$

In addition to the attack strategy and defense strategy, the constructed datasets are also determined by the target victim program (Figure 1), *i.e.*, the victim's memory traces. Since our goal is to analyze the effectiveness of the attack strategy or defense strategy, we can fix the victim behavior during cache modeling. In our experiments, without loss of generality, we model the victim program as accessing a random cache block within a critical region at a fixed speed. It is also possible to use the memory traces of the victim's actual program, *e.g.*, AES, as the victim's behavior to generate datasets.

The second step is to train the neural network and evaluate its accuracy. This simulates the process of side-channel analysis and secret recovery. In the training phase, a neural network model is built over the training dataset D^{train} to reveal the relationship between the side-channel information and secrets. In the inference phase, the testing dataset D^{test} is used to check how accurate this model is. The evaluation results are used as the metric of the adversary's capability. This represents how much critical information the adversary can obtain correctly from his observations. A weak adversary will have a low accuracy as his observation has little relationship with the secrets. This metric can also be used to quantify the effectiveness of a defense solution. An effective solution tries to maximize the loss function and minimize the correlation between input and output tensors in the model training phase. Then we will observe a low prediction accuracy in the model inference phase.

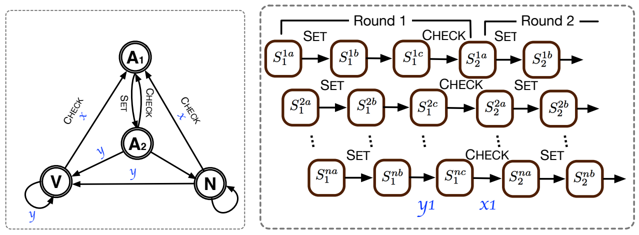
In Sec. 4 we show how to model cache activities under different types of attacks and defenses for datasets generation. In Sec. 5 we show how to design, train and evaluate the neural network.

4 DATASET CONSTRUCTION

We first describe an abstract state machine model for the side-channel operations. Then we show how models of specific attacks and defenses can be derived from the abstract one.

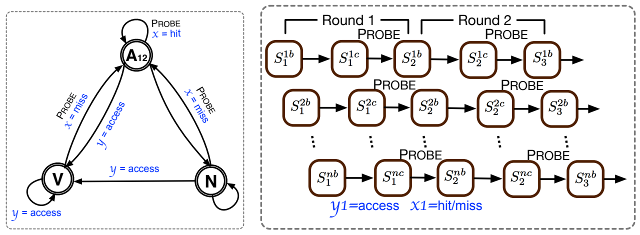
4.1 An Abstract Model

Modeling a single cache object . Figure 2a shows the state machine of an "object" in cache modeling. An "object" can be an entire cache set when the adversary acts on the cache set to retrieve information (*e.g.*, PRIME-PROBE, EVICT-TIME, PRIME-ABORT). It can also be a memory block when the adversary only needs to operate on individual memory blocks. (*e.g.*, FLUSH-RELOAD, FLUSH-FLUSH).



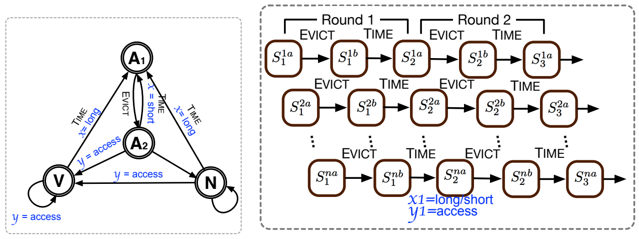
(a) Single object (b) Multiple objects

Figure 2: The abstract state machine.



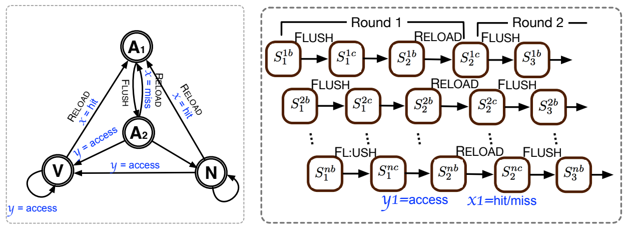
(a) Single object (b) Multiple objects

Figure 3: Prime-Probe state machine.



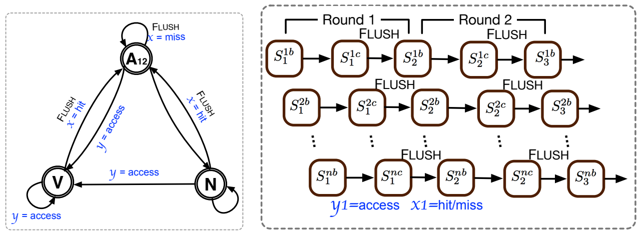
(a) Single object (b) Multiple objects

Figure 4: Evict-Time state machine.



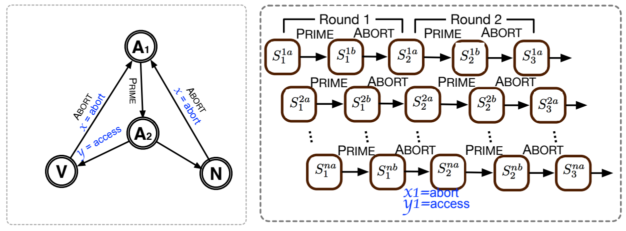
(a) Single object (b) Multiple objects

Figure 5: Flush-Reload state machine.



(a) Single object (b) Multiple objects

Figure 6: Flush-Flush state machine.



(a) Single object (b) Multiple objects

Figure 7: Prime-Abort state machine.

The cache object is modeled as one of these states based on the Set and Check operations:

- $\mathbf{A_1}$: the adversary just finishes Check on this object.
- $\mathbf{A_2}$: the adversary just finishes Set on this object.
- **V**: this object is accessed by the victim between Set and Check.
- **N**: this object is accessed between Set and Check, but not by the victim.

In Figure 2a we show the transition rules with the source/destination states. We also show the adversary's observations from Check (x in blue) and victim's activities (y in blue) collected during the state transitions to form the datasets. Basically the adversary's activities (*i.e.*, Set and Check), the victim's activities (y), and other processes' activities (not marked in the figures) can cause state transitions. We describe each possible transition below. Algorithm 1 shows the algorithm of these rules.

- $\mathbf{A_1} \mapsto \mathbf{A_2}$: in state $\mathbf{A_1}$, if the adversary conducts Set, the object jumps to state $\mathbf{A_2}$. Note that we do not consider the transitions $\mathbf{A_1} \mapsto \mathbf{V}$ or $\mathbf{A_1} \mapsto \mathbf{N}$ because we assume that the adversary conducts Set immediately after Check.
- $\mathbf{A_2} \mapsto \mathbf{V}$: in state $\mathbf{A_2}$, if the victim accesses this object, the object jumps to state **V**. This access is recorded into the dataset.
- $\mathbf{A_2} \mapsto \mathbf{N}$: in state $\mathbf{A_2}$, if a process other than the victim or the adversary accesses this object, the object jumps to state **N**.
- $\mathbf{N} \mapsto \mathbf{V}$: in state **N**, if the victim accesses this object, the object jumps to state **V**. This access is collected into the dataset.
- $\mathbf{N} \mapsto \mathbf{N}$: in state **N**, if a process other than the victim or the adversary accesses this object, the object stays in state **N**.
- $\mathbf{V} \mapsto \mathbf{V}$: in state **V**, any accesses to this object makes it stay in state **V**. This is because state **V** denotes that this object has already been accessed by the victim between Set and Check. So other events will not change the state except Check. If it is a victim's access, it is recorded into the dataset.
- $\mathbf{V} \mapsto \mathbf{A_1}$: in state **V**, when the adversary conducts Check, the object jumps back to state $\mathbf{A_1}$ and the adversary captures some information caused by the victim's activity. Such information is recorded into the dataset.

Algorithm 1: Modeling a single cache object

```
function StateMachineTransition(State, Activity)
    if State = $A_1$ then
        if Activity = adversary:SET then
            NewState = $A_2$
            Observation = False
        end
    end
    if State = $A_2$ then
        if Activity = adversary:CHECK then
            NewState = $A_1$
            Observation = False
        end
        if Activity = victim:access then
            NewState = V
            Observation = False
        end
        if Activity = other:access then
            NewState = N
            Observation = False
        end
    end
    if State = V then
        if Activity = victim:access|other:access then
            NewState = V
            Observation = False
        end
        if Activity = adversary:CHECK then
            NewState = $A_1$
            Observation = True
        end
    end
    if State = N then
        if Activity = victim:access then
            NewState = V
            Observation = False
        end
        if Activity = other:access then
            NewState = N
            Observation = False
        end
        if Activity = adversary:CHECK then
            NewState = $A_1$
            Observation = True
        end
    end
    return {NewState, Observation}
end
```

- $\mathbf{N} \mapsto \mathbf{A_1}$: in state **N**, when the adversary conducts CHECK, the object jumps back to state $\mathbf{A_1}$ and the adversary captures some information caused by processes other than the victim. Such information is recorded into the dataset and the adversary cannot distinguish if this activity is caused by the victim or not.
- $\mathbf{A_2} \mapsto \mathbf{A_1}$: in state $\mathbf{A_2}$, when the adversary conducts CHECK, the object jumps to state $\mathbf{A_1}$. The adversary does not capture any information since no activities occur between SET and CHECK.

Modeling multiple cache objects. To consider all the activities within the critical cache region, all the objects in this region are included, with each one modeled as the state machine in Figure 2a. Figure 2b shows how n cache objects are modeled within the critical cache region under a side-channel attack. S_j^{ia}, S_j^{ib} and S_j^{ic} denote that the ith object is in different stages in the jth round. Basically for each round the adversary conducts SET on each object, waits for a certain time (could be zero) and then conducts CHECK on each object. The victim's memory activity is modeled as randomly accessing one critical cache block in the critical region at a constant speed. The adversary's observation from the CHECK operation, and the victim's memory activity form a data sample. The adversary repeats these operations and a dataset is thus constructed. Algorithm 2 describes this process.

Algorithm 2: Modeling multiple cache objects

```
function DataSetGeneration($N_{object}$, $N_{round}$)
    for j = 1 ...... $N_{round}$ do
        for i = 1 ...... $N_{object}$ do
            $\{S_i^{jb}, -\}$ = StateMachineTransition($S_i^{ja}$, adversary:SET)
            while adversary is waiting do
                if victim's access to object then
                    $\{S_i^{jb}, -\}$ = StateMachineTransition($S_i^{jb}$, victim:access)
                    $x_j^i = 1$
                end
                if other access to this object then
                    $\{S_i^{jb}, -\}$ = StateMachineTransition($S_i^{jb}$, other:access)
                end
            end
            $S_i^{jc} = S_i^{jb}$
            $\{S_{i+1}^{ja}$, Observation$\}$ = StateMachineTransition($S_i^{jc}$, adversary:CHECK)
            if Observation = True then
                $y_j^i = 1$
            end
        end
    end
    return $\{x_i = [x_i^1, x_i^2, \ldots, x_i^n], y_i = [y_i^1, y_i^2, \ldots, y_i^n]\}_{i=1}^N$
end
```

4.2 Modeling Specific Attacks

The models of specific side-channel attacks can be derived from the abstract model with minor changes.

Contention-based attacks. In PRIME-PROBE and EVICT-TIME attacks, a cache set is considered as an object. So we model each critical cache set as a state machine.

PRIME-PROBE attack: the modified state machine of a cache set is shown in Figure 3a. As we described in Sec. 2.1, the PROBE operation is the PRIME operation for the next round. so the adversary just needs to conduct one PROBE operation continuously. As a result, the states $\mathbf{A_1}$ and $\mathbf{A_2}$ are combined into a new state $\mathbf{A_{12}}$. For the dataset, $\mathcal{X}$ denotes if the PROBE encounters a cache hit or miss. $\mathcal{X}$ is a hit for the transition $\mathbf{A_{12}} \mapsto \mathbf{A_{12}}$, as no other activities happen in this cache set during the interval and the adversary's cache blocks are not evicted out. $\mathcal{X}$ is a miss for transition $\mathbf{V} \mapsto \mathbf{A_{12}}$ or $\mathbf{N} \mapsto \mathbf{A_{12}}$ as the victim or other processes access this cache set and evict the adversary's data out. $\mathcal{Y}$ denotes if the victim accesses this set during the interval. It happens for transitions $\mathbf{A_{12}} \mapsto \mathbf{V}$, $\mathbf{N} \mapsto \mathbf{V}$ and $\mathbf{V} \mapsto \mathbf{V}$.

One change is also required to the model of multiple cache objects (Figure 3b). Since the SET and CHECK are combined into one operation, we remove the state S_j^{ia} and the SET operation for each round j and each object i. One CHECK (or PROBE) operation is enough in each round. This operation collects the adversary's observations, as well as sets the cache state for the next round.

EVICT-TIME attack: Figure 4a shows the state machine of a cache set. Since the EVICT and TIME are two different operations, the $\mathbf{A_1}$ and $\mathbf{A_2}$ states are separated. $\mathcal{X}$ denotes if the measured time of the interval is long or short[1]: a short time means there is no memory access to this set ($\mathbf{A_2} \mapsto \mathbf{A_1}$) and a long time means this

[1]Typically the interval in the EVICT-TIME attack is one encryption block, and the adversary collects a large number of measurements to calculate the average time. Our model just abstracts the interval as one memory access to ignore the average effort. This is a special case for EVICT-TIME where the adversary's capability is maximized. It still represents the essential feature of EVICT-TIME attack

set is accessed during this interval ($\mathbf{V} \mapsto \mathbf{A_1}$ or $\mathbf{N} \mapsto \mathbf{A_1}$). Similarly, $\mathcal{Y}$ denotes a victim's access to this set.

Figure 4b shows the model of multiple objects in Evict-Time. Different from the abstract one, there are no wait intervals. After the adversary sets the cache state, he immediately measures the victim's access time. Thus the victim's activity and the adversary's observation occur at the same phase.

Reuse-based attacks. In Flush-Reload and Flush-Flush attacks, a cache block is treated as an object.

Flush-Reload attack: as shown in Figure 5a, the state machine in Flush-Reload is quite similar to Evict-Time. The difference is that an access to this cache block will cause a cache hit for the adversary's observation $\mathcal{X}$, as this cache block is brought to the cache by the victim. If the cache block is not accessed during the interval, then $\mathcal{X}$ is a cache miss. Figure 5b shows how we consider multiple cache blocks. The model is exactly the same as the abstract one, with Set as Flush and Check as Reload.

Flush-Flush attack: similar to Prime-Probe, this attack uses just one operation Flush to both Set and Check the target memory block. So the state machine is also similar (Figure 6a). $\mathcal{X}$ is a cache hit when there is an access to this block and a cache miss when there is not. The model of multiple cache blocks is also similar to that of Prime-Probe.

Abort-based attacks. In Prime-Abort attacks, a cache set is considered as an object.

Prime-Abort attack: different from Prime-Probe attack, this attack checks the cache set states by observing an abort signal. So the state machine (Figure 7a) is also different: first, states $\mathbf{A_1}$ and $\mathbf{A_2}$ are separate because the adversary has two different operations. Second, there is no transition $\mathbf{A_2} \mapsto \mathbf{A_1}$ because once the adversary conducts the Prime operataion, it will wait until one process evicts his data out of the cache set and triggers an abort. Third, access to any set inside the critical region can trigger the abort to the adversary. So there are no transitions $\mathbf{V} \mapsto \mathbf{V}$, $\mathbf{N} \mapsto \mathbf{V}$ or $\mathbf{N} \mapsto \mathbf{N}$. We will compare the efficiency of Prime-Probe attack and Prime-Abort attack caused by these differences in Sec. 6.

Figure 7b shows the model of multiple cache sets. The victim's access triggers the abort signal in the adversary's observation, so they happen in the same phase. This is similar to Evict-Time.

4.3 Modeling Defense Solutions

In addition to the adversary's behaviors, the cache system and defense solutions that can affect the attack effects also need to be modeled. We show how to model the cache security policy P in the defense strategy $\lambda^{defense}$. We consider isolation and randomization, as introduced in Sec. 2.2.

Isolation. Isolating the adversary's and victim's cache activities indicates that there are no transitions between states $\mathbf{A_1}$ ($\mathbf{A_2}$), $\mathbf{V}$ and $\mathbf{N}$. So in Figure 2a transitions $\mathbf{A_2} \mapsto \mathbf{V}$, $\mathbf{A_2} \mapsto \mathbf{N}$, $\mathbf{V} \mapsto \mathbf{A_1}$, $\mathbf{N} \mapsto \mathbf{A_1}$, and $\mathbf{N} \mapsto \mathbf{V}$ are removed. By doing so, each object will fall into one of the three possibilities: restricted to states $\mathbf{A_1}$ and $\mathbf{A_2}$, restricted to state $\mathbf{V}$, and restricted to state $\mathbf{N}$. The specific attack models in Figures 3a – 7a can be modified in a similar way.

Randomization. We use two examples to illustrate how randomization policies are modeled in the cache state machines.

Adding random cache activities: the idea is to add random fake noise to the adversary's observation. For attacks that target the entire cache sets (Prime-Probe, Evict-Time, Prime-Abort), a random cache set is selected frequently and dynamically, and one block from this set is evicted out of the cache [55]. To model such scheme, state $\mathbf{N}$ and the corresponding transitions are used. A random object is selected. If the object is in state $\mathbf{A_2}$ (Figure 2a), a transition $\mathbf{A_2} \mapsto \mathbf{N}$ is generated to denote random eviction of the adversary's block. Later on this cache set will go back to $\mathbf{A_1}$ and the adversary observes an activity with the Check operation. However, he cannot distinguish whether the source of this transition is $\mathbf{V}$ or $\mathbf{N}$. This uncertainty can lower the adversary's prediction accuracy. If the selected object is in state $\mathbf{V}$, then the object will still stay in state $\mathbf{V}$ after evicting out the victim's block in this set. This will not add noise to the adversary's observation.

For attacks that target individual cache blocks (Flush-Reload, Flush-Flush), more processes that share the same cache blocks with the adversary and victim [58] can be added. Similarly, state $\mathbf{N}$ can be used to model the effects of other noisy processes that the adversary is not interested in. The transition $\mathbf{A_2} \mapsto \mathbf{N}$ can be viewed as generating a false alarm to the adversary's observation.

Randomizing memory-to-cache mapping: this scheme is effective for contention-based and abort-based attacks, but not for reuse-based attacks. A mapping table is kept for each process to map the memory addresses to cache sets. Two random sets are dynamically swapped in each mapping table to randomize the memory-to-cache mapping. For each swap the data that belong to that process in these two sets also need to be evicted out. By doing so each process's memory-to-cache mapping is random and different. When the adversary captures a cache set access, he can not get its memory address. This can prevent information leakage.

5 DNN TRAINING AND INFERENCE

5.1 Dataset Processing

The first step is to process the dataset generated from Sec. 4. We consider a critical cache region with n objects. These objects store the victim's secrets. A dataset with S samples is generated. So the cache activities for S Set-Check rounds need to be modeled. In each round one sample is collected.

The input of a sample can be formatted as an n-dimensional vector $x_i = [x_i^1, x_i^2, .., , x_i^n]$, to denote if the adversary observes an activity in each object during round i. If $x_i^j = 1$, then the adversary observes an activity in the jth object in round i. For Prime-Probe attack, this is a cache miss in cache set j. For Flush-Reload and Flush-Flush attacks, this is a cache hit in cache block j. For Prime-Abort attack, this is an abort event to the adversary's program. If $x_i^j = 0$, then the adversary does not observe any activity in object j.

The output of a sample can also be formatted to an n-dimensional vector $y_i = [y_i^1, y_i^2, .., , y_i^n]$, to denote the access event of each object by the victim during round i. If $y_i^j = 0$, it means the jth object is not accessed by the victim in this round. If $y_i^j = k$, it means that the jth object is accessed by the victim in this round, and it is the kth access of the victim. For instance, considering a 5-object cache region. If $y_0 = [0, 0, 0, 0, 0]$, it indicates that the victim does not access the region in round 0. If $y_1 = [0, 0, 1, 0, 0]$, it means that the victim only accesses the third object in round 1. If $y_2 = [0, 2, 0, 0, 1]$, it indicates that in round 2, the victim accesses the fifth object first, and then the second object. Note that in this way the output also records the victim's access order. The adversary's prediction accuracy of the objects accessed and their access order in each round is measured.

5.2 Training

Given the training dataset, a neural network is trained to establish the relationship between the adversary's observation and victim's access. In this paper we choose a multilayer perceptron network (Sec. 2.3) with three hidden layers. These hidden layers have the same shape as the input/output (*i.e.*, n-dimensional vector), and the `ReLU` activation function is applied to each layer following a linear transformation (Equation 1). The `softmax` function is applied to the final layer to get the output. Our evaluations in Sec. 6 show that this network architecture is already powerful enough to reveal information leakage in various side-channel attacks.

The cross entropy is chosen as the loss function. This loss function denotes how much the adversary's guessing is wrong on the training dataset. Stochastic gradient descent is exploited to find the optimal parameters that can minimize this loss.

5.3 Inference

After obtaining the trained model with optimized parameters, the prediction accuracy over the testing dataset is measured to quantify the adversary's capability or the effectiveness of the defense solution. The prediction accuracy is defined as the ratio of the victim's accesses that the adversary can correctly predict over the total number of victim's critical accesses.

Specifically, for one sample $\{x_i, y_i\}$ collected at round i, we assume that the victim makes k memory accesses. We denote that these k memory accesses touch the objects $l_1, l_2, ..., l_k$ in chronological order. According to the data formation rule in Sec. 5.1, for the ground-truth label y_i, $y_i^{l_1} = 1, y_i^{l_2} = 2, ..., y_i^{l_k} = k$, and the rest of the elements are all zero.

Then the predicted label is calculated as $\hat{y_i} = f_{\theta^*}(x_i)$, which is also an n-dimensional vector. The top-k elements in $\hat{y_i}$ are selected [2]. The values of these elements are changed to $1, 2, ..., k$ based on the order discovered by the Check operation, while the other elements in $\hat{y_i}$ are set to zero. The ratio of the top-k elements in $\hat{y_i}$ that match the values of the corresponding elements in y_i are checked as the accuracy of this sample. The total accuracy is the average accuracy of all the samples in consideration.

We use a 5-object cache region as an example to illustrate the accuracy prediction process. Assume the ground-truth label $y_0 = [0, 0, 1, 0, 0]$. If $\hat{y_0} = [0.01, 0.02, 0.93, 0.03, 0.01]$, then the top-1 element is the third one, and the output is converted to $\hat{y_0} = [0, 0, 1, 0, 0]$. This exactly matches y_0, and the accuracy of this sample is 100%. If $\hat{y_0} = [0.01, 0.02, 0.45, 0.51, 0.01]$, then it is converted to $\hat{y_0} = [0, 0, 0, 1, 0]$, and the accuracy becomes 0% since the model mis-predicts the victim's access from the third object to the fourth object. Assume the victim has two accesses in one round, and the ground-truth label now becomes $y_1 = [0, 0, 1, 2, 0]$. If the predicted output is $\hat{y_1} = [0.01, 0.02, 0.55, 0.41, 0.01]$, then the top-2 elements are the third and fourth ones, and the output is converted to $\hat{y_1} = [0, 0, 1, 2, 0]$. This prediction gives a 100% accuracy. If $\hat{y_1} = [0.01, 0.02, 0.48, 0.22, 0.27]$, then the output is post-processed as $\hat{y_1} = [0, 0, 1, 0, 2]$, and the prediction accuracy becomes 50%, since it only correctly predicts one access out of two.

[2]We make the assumption that the adversary knows the number of victim's memory accesses in each attack round. This maximizes the adversary's capability. He can deduce the number of victim's accesses in each period by checking the attack time and knowledge of victim program and runtime characteristics.

6 EVALUATION

We write python code to build the state machines and construct the datasets for different attacks and defenses. Specifically in this evaluation, we consider a critical cache region of 64 sets with 16 ways. Note this region is not necessarily the whole cache: it can be the region that stores the victim's critical information, e.g., cryptographic lookup tables. We assume that the victim's critical data are stored in one cache block in each of the 64 sets (for a large conventional set-associative cache, it is very common that each critical block is mapped to a different cache set [34]). We model multiple attack rounds as described in Sec. 4.1. In each round we record the adversary's observation $\mathcal{X}$ and victim's access trace $\mathcal{Y}$.

We implement the deep neural network in MxNet (version 0.11.0) [9]. We train the multilayer perception neural network using stochastic gradient descent with a learning rate of 0.01, and a batch size of 64. We have 6,400 samples as training dataset by default and 640 samples as testing dataset. We measure the prediction accuracy over the testing dataset.

6.1 Attack Strategies

We first consider the effects of the attack strategies, λ^{attack}, which were detailed in Sec. 2.1. To do so, we change and compare different attack strategies on a conventional cache without any defenses.

Attack techniques. First we compare different attacks within the same categories, *i.e.*, Prime-Probe VS. Evict-Time in the contention-based category, and Flush-Reload VS. Flush-Flush in the reuse-based category. We describe the reuse-based category as an example below. The contention-based category has the same conclusion. For the state machines, the difference between Figures 6a and 5a is that in Figure 5a there are two states $\mathbf{A_1}$ and $\mathbf{A_2}$ while in Figure 6a there is only one $\mathbf{A_{12}}$. If there are no activities happening between $\mathbf{A_1}$ and $\mathbf{A_2}$ in Figure 5a, we can simply combine them into one, and convert the state machine into Figure 6a. So the attack techniques within the same category should have the same effect. If there are intervals between two rounds, i.e., between Check in one round and Set in the next round, and the victim has actions during these intervals (we do not show such transition in Figure 5a), then the adversary cannot capture these actions and will have a lower prediction accuracy.

Figure 8 shows the comparisons of prediction accuracy in these two attacks. In this figure, the x-axis is the number of training epochs and y-axis is the accuracy. We assume that there is a victim access between each Set and Check round. The black line shows the prediction accuracy of the Flush-Flush attack. The adversary can achieve 100% accuracy after enough training epochs, indicating that he can easily figure out all the victim's accesses with the correct order. The remaining lines show the adversary's accuracy in Flush-Reload with different interval sizes between each round. The red

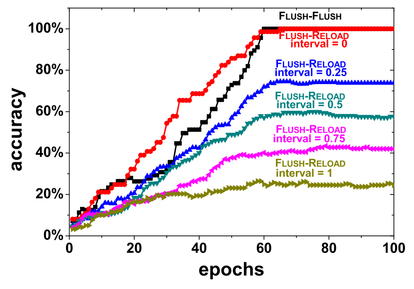


Figure 8: Comparison between attacks in the same category

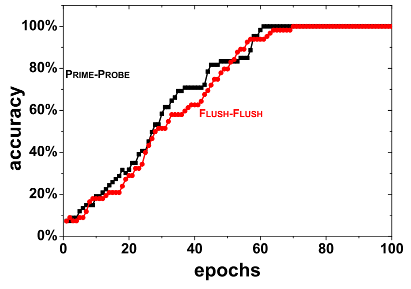


Figure 9: Comparing Contention-based v.s. Reuse-based attacks

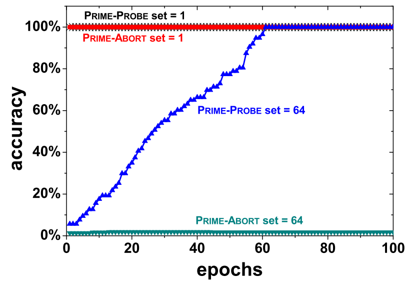
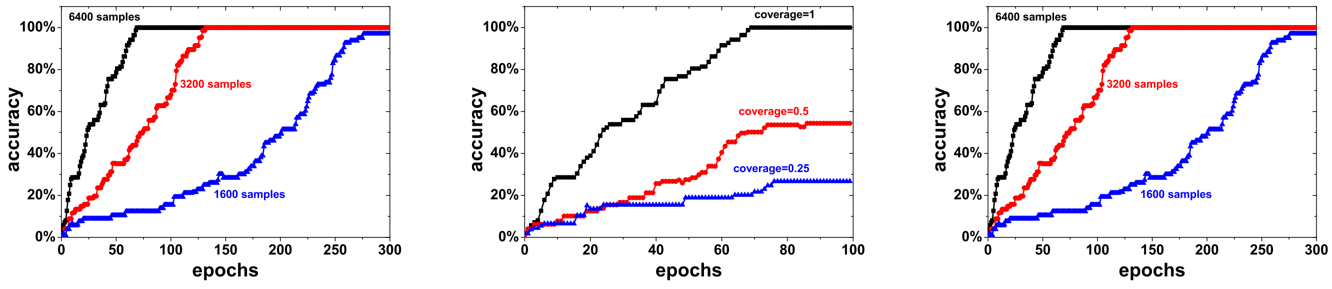


Figure 10: Comparing Contention-based v.s. Abort-based attacks



(a) Attack speeds (b) Attack coverages (c) Attack durations

Figure 11: Prediction accuracies with different attack factors

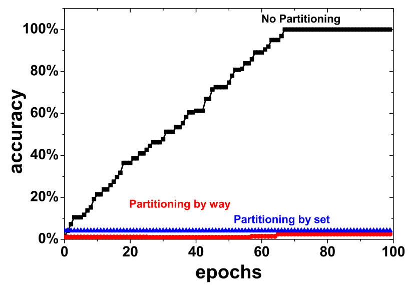


Figure 12: Prediction accuracies for cache partitioning

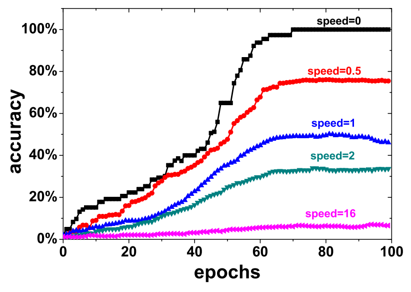


Figure 13: Prediction accuracies for random eviction

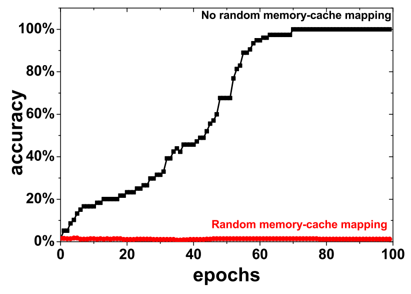


Figure 14: Prediction accuracies for random memory-cache mapping

line (interval = 0) is the case that the FLUSH operation immediately follows the RELOAD operation in the previous round. We observe the attack effect is the same as the FLUSH-FLUSH. The blue line (interval = 0.25) is the case that the RELOAD − FLUSH interval is 1/4 of the FLUSH − RELOAD interval. The adversary cannot achieve 100% accuracy as he cannot capture the victim's access between the RELOAD − FLUSH interval. The larger the interval is, the lower the prediction accuracy the adversary gets. The comparison between PRIME-PROBE and EVICT-TIME has similar results (not shown).

To conclude, if there is no interval between CHECK and SET operations, the attacks in each category are essentially the same. Note that we abstract the SET and CHECK operations and treat them the same in all the attacks. In practice, conducting the CHECK and SET operations are different for different attacks. This makes these attacks different in efficiency. For instance, flushing a cache block is faster than reloading the cache block. So FLUSH-FLUSH is claimed to be faster than FLUSH-RELOAD [17]. In EVICT-TIME, the adversary needs to conduct a large quantity of encryptions and calculate the average time to obtain the results of the TIME operation. So it also takes longer time than PRIME-PROBE attack.

Second, we compare contention-based and reuse-based attacks. Figure 9 shows the results of the PRIME-PROBE attack (black line) and FLUSH-FLUSH attack (red line). We also assume that there is exactly one victim's access between each SET and CHECK round. We observe that these two attacks have the same effects, as they have the same state machine modeling.

We also abstract the SET and CHECK operations in the two attacks. If we consider their implementation, the attack effects will be different: the Flush-Flush attack only needs to issue one instruction for each operation, while the Prime-Probe attack needs to access the entire cache set to complete one operation. So in practice, Flush-Flush is more efficient than Prime-Probe. But it also has an extra requirement: memory page sharing.

Third, we consider the comparison between Prime-Probe and Prime-Abort. The difference between these two attacks is the Check operation: in Prime-Probe, the adversary scans each entire cache set to identify the victim's footprints; in Prime-Abort, the adversary just waits for the abort signal to capture information – the victim's access to any critical cache set can cause the adversary's abort, and he cannot distinguish which set is touched by the victim.

Figure 10 shows the results of the two attacks. We consider two cases: the number of critical cache sets is 1 and 64. From this figure we can observe that when there is only one target cache set, then Prime-Probe and Prime-Abort can both achieve 100% accuracy. This is straightforward as the adversary can accurately capture the victim's access to this set by either Probe or Abort. However, when the number of critical cache sets is 64, the Prime-Probe adversary can achieve 100% accuracy after some training epochs, while the Prime-Abort adversary has a very low prediction accuracy, as he cannot tell which set is accessed by the victim when he gets an abort signal. So when the victim's secrets are stored in multiple cache sets, the Prime-Probe attack is more efficient as it can get more fine-grained information about each set. In practice, the Prime-Abort is easier and more practical to implement as it does not need to do anything for the Check operation, while the Prime-Probe needs to scan the entire cache set.

Attack factors. We consider other factors that can affect the attack effects. We use the Prime-Probe attack as an example. Other attacks have similar conclusions.

First we consider the attack speed. This factor can significantly affect the feasibility of attacks: the faster the adversary can conduct Set-Check, the more fine-grained information he can infer from the cache states. A lot of techniques have been proposed to increase the adversary's speed relative to the victim's accesses [19, 57].

Figure 11a shows the model prediction accuracy when the adversary conducts the Prime-Probe at different speeds. We set the relative speed of Prime-Probe to the victim's access as 1 (black line: the victim has one access within one Prime-Probe round), 0.5 (red line: the victim has two accesses within one round) and 0.25 (blue line: the victim has four accesses within one round). We observe that lower relative speed leads to lower accuracy. When the relative speed is 1, the adversary can easily figure out all the accesses in the correct order. When the relative speed is smaller, the victim issues more cache accesses within one round. This decreases the adversary's accuracy because the adversary cannot recognize the relative order of these two accesses. Such uncertainty causes low prediction accuracy.

Second, we consider attack coverage, defined as the ratio of the critical objects that the adversary can cover. The higher coverage the adversary can achieve, the more information he can obtain.

Figure 11b shows the prediction accuracy under different coverages for the Prime-Probe attack. The baseline is the case where the adversary can affect all the objects (coverage=1), and the neural network model can achieve 100% accuracy. When the coverage is reduced to 0.5, the adversary can only interfere with half of the victim's memory accesses, so the accuracy decreases to around 50%. When the coverage is 0.25, the accuracy drops to 25%. A coverage of $1/n$ gives an accuracy of approximately $1/n \times 100\%$.

Third we consider the attack duration. The attack duration is determined by the number of attack rounds (*i.e.*, number of training samples) required to identify the side-channel relationship. The adversary expects a short attack duration, which might enable him to collect enough samples for training before the victim makes any changes or deploys defenses.

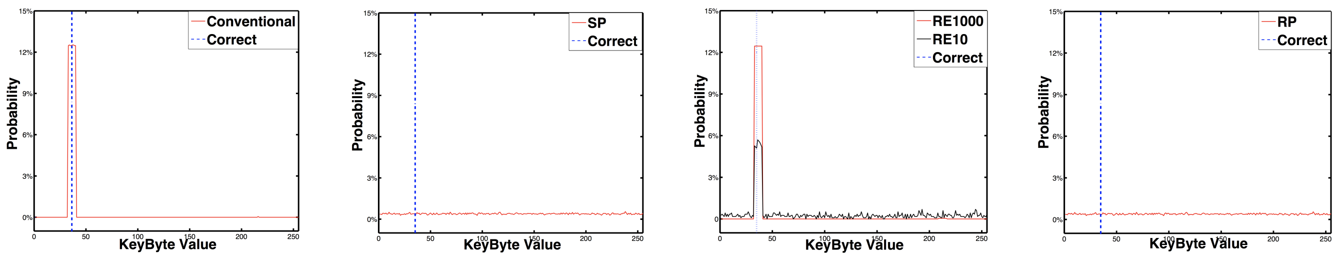
Figure 11c shows the prediction accuracy of Prime-Probe attack with different numbers of training samples. From this figure we observe that reducing the number of samples does not affect the accuracy. It only takes longer time to analyze these samples to generate the model. This indicates that the neural network is powerful to reveal the side-channel relationship even with a small number of training samples, at the cost of longer offline analysis time.

6.2 Defense Strategies

Next we show how to use the neural network approach to quantify the effects of different defense solutions. We evaluate two categories of defenses: isolation and randomization. We maximize the adversary's capability, *i.e.*, the relative speed of Set-Check to the victim's access is 1, the cache region coverage is 1, and the adversary can collect arbitrary number of training samples.

Isolation. We use the Prime-Probe attack as an example to measure the effectiveness of isolation. We can physically partition the CPU cache into two zones (by sets or by ways). One zone is allocated to the victim and the other is allocated to the adversary. Other isolation approaches on other attacks (*e.g.*, disabling page sharing to defeat Flush-Reload or Flush-Flush attacks) have similar results. Figure 12 shows the model prediction accuracy with no cache partitioning (black line), partitioning by ways (red line) and partitioning by sets (blue line). We can see that both of the two partitioning policies can reduce the prediction accuracy to close to zero. When the cache is partitioned by sets, the adversary cannot observe any cache misses as the victim cannot interfere with the adversary's data. When the cache is partitioned by ways, the adversary are always cache misses caused by himself as the number of ways allocated to him is reduced. In either case the adversary's observation is totally independent of the victim's activities.

Randomization. We consider two randomization approaches, as introduced in Sec. 2.2. The first one is to add random cache activities to the adversary's observation. We use the random eviction policy in Prime-Probe attack as an example, where a cache block in a random cache set is periodically selected to be evicted out. Other policies of adding random cache activities for other attacks (*e.g.*, adding noisy processes in reuse-based attacks) are essentially the same and have similar results. Figure 13 shows the model prediction accuracy at different random eviction speeds (*i.e.*, how many cache blocks are evicted during one Prime-Probe round). We can observe that a faster eviction speed can significantly reduce the model prediction accuracy. When the eviction speed is 16, the prediction accuracy is 0.1, indicating that the adversary's observation has a very weak dependency on the victim's activities.



(a) Conventional cache (b) Statically partitioned cache (c) Random eviction cache (d) Random permutation cache

Figure 15: Simulation results of Prime-Probe attacks [55] on different types of secure caches

The second approach is to randomize the memory-to-cache mappings to defeat contention-based or abort-based side-channel attacks (Random Permutation [51], NewCache [33, 49]). We use Prime-Probe attack as an example, and randomly and dynamically change the mapping from memory addresses to hardware-remapped cache sets for the adversary's and victim's processes. Figure 14 shows the accuracy of the adversary's prediction of the victim's memory traces, from his observation, with and without random memory-to-cache mapping mechanisms. When the random mapping is enabled, the adversary's observation has no relationship with the victim's memory trace due to the dynamic and random mapping. The prediction accuracy is very low, close to random guessing.

7 METHODOLOGY VALIDATION

To validate the deep-learning based approach, we also conducted actual attacks to confirm some results of our side-channel analysis from Sec. 6. Here we only show the validation and comparison about different defense strategies.

Recall that in Sec. 6.2 we show the quantification results of Prime-Probe attack under different defense policies as a representative (Figures 12, 13 and 14). So we adopt the results of actual Prime-Probe attacks from [55] for comparison. Figure 15 shows the simulation results from the gem5 platform, a popular cycle-accurate simulator. The victim runs an AES-128 encryption program (i.e., the key length is 16 bytes). The adversary runs the Prime-Probe attack [36] to steal the encryption key. The two programs share the same level 1 cache, and different defense policies are implemented.

Figure 15 shows the recovery results of the first key byte. The values of the other 15 key bytes can be leaked in a similar way. The x-axis shows the possible values of this key byte (0 − 255). The y-axis is the probability of the key-byte value from the attack. Thus the solid red line is the probability distribution of key-byte values the adversary obtains. We also show the correct key value as dotted blue lines in the figures.

When no defense is implemented (Figure 15a), 8 values (out of 256) are more likely than others (more than 10% probabilities from the attack results), and the correct key-byte value is among them. So the adversary is able to narrow down the key-byte scope, and the attack on this conventional cache succeeds. When the cache is statically partitioned (SP) between the victim and the adversary (Figure 15b), the attack does not produce any distinguished candidate keys. Thus the attack on partitioned cache fails. We achieve the same conclusion for the random permutation policy (Figure 15d). For random eviction policy (Figure 15c), we consider two random eviction speeds: RE1000 (a random cache block is evicted every 1000 memory accesses - in red) and RE10 (a block is evicted every 10 memory accesses - in black). We can see that random eviction cache is still vulnerable to the Prime-Probe attack. A faster random eviction speed leaks less information to the adversary. Comparing Figures 12, 13, 14 and 15, we confirm that the results from the neural network analysis are consistent with the results from actual attacks.

8 RELATED WORK

Different models and methods were proposed to evaluate side-channel attacks. They can be mainly classified into three categories.

The first category proposes new metrics. Some work used mutual information and guessing entropy to evaluate the feasibility of key-recovery [45], estimate the upper bound of information leakage [26], improve the adversary's strategy [25], and evaluate the security of defenses [55]. Cañones et al. [8] built Mealy machine and adopted Markov chain to quantify the amount of information absorbed by the cache, and the amount of information extracted by the adversary. Some work adopted the success rate metric [45] to calculate the probability that the adversary can detect a memory access [12] or determine the best attack strategy [40]. A more popular metric is to calculate the correlation between the victim's execution and adversary's observation. Several studies followed this direction, *e.g.*, time-driven analytic model [47], Side-channel Vulnerability Factor (SVF) [10], timing SVF [5], Cache Side-channel Vulnerability (CSV) [54].

The second category uses information flow tracking to capture side-channel information leakage. Porras and Kemmerer designed the technique of covert flow trees to systematically detect and identify covert channels between processes [38]. New hardware languages were designed which use static information flow tracking to verify side-channel leakage in hardware design [14, 53]. He and Lee built an analytic probabilistic information flow tracking model to evaluate side-channel vulnerabilities of different caches [20].

The third category includes formal verification of side channels. Svenningsson and Sands [46] built models of timing side-channel leakage from the program code level. Bacelar et al. [3] adopted the self-composition technique to verify the non-interference property of cryptographic software, thus to evaluate the effectiveness of side-channel countermeasures.

Different from these prior studies, we adopt a neural network to quantify side channels. Compared to other methods, our approach has the following advantages: (1) metrics proposed by prior work assume that the adversary's observation and victim's execution have a linear relationship [5, 10, 47, 54], which greatly limits their applicability. In contrast, our neural network model can discover non-linear relationships and can be applied to evaluate more types of side channels; (2) some prior work need to manually deduce the relationship between the victim and the adversary based on prior knowledge about how the adversary steals the secrets [25, 40]. In contrast, our approach does not need such prior knowledge. We only need to feed the adversary and victim's behaviors into the neural network, and the network will automatically reveal the relationship for us. This significantly reduces the difficulty of side channel analysis; (3) some approaches are designed for one specific type of cache side-channel attacks [5, 10, 47, 54, 55]. Some approach can only be used for evaluating the security mechanisms of cache architectures [55] or cache replacement policies [8]. Our methodology is flexible to quantify different attack strategies and types, as well as defense strategies.

9 CONCLUSION

This work proposes a novel deep learning based methodology to analyze cache side channels. We treat the side-channel attack as a neural network with the adversary's observations as the inputs and the victim's activities as the outputs. This neural network can automatically discover the side-channel relationship, and evaluate the prediction accuracy. We build state machines to model different side-channel attacks and defenses, and use model prediction accuracy as a metric to quantify the effectiveness of these strategies.

The essence of side-channel attacks is to *learn* and *predict* the critical information from the observed side-channel information. This agrees with the goal of deep learning. So this method is not limited to cache-based side channel analysis. Future work include extending this method to other types of side channel evaluation, and discovering new side-channel attacks using deep learning.

REFERENCES

[1] [n. d.]. Kernel Samepage Merging. http://www.linux-kvm.org/page/KSM.

[2] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *Intl. Conf. on Cryptographic Hardware and Embedded Systems*.

[3] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. 2013. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.* (2013).

[4] Daniel J. Bernstein. 2005. *Cache-timing attacks on AES.* Technical Report.

[5] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. 2012. Hardware Prefetchers Leak: A Revisit of SVF for Cache-Timing Attacks. In *Hardware and Architectural Support for Security and Privacy*.

[6] Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision Timing Attacks against AES. In *Lecture Notes in Computer Science series 4249*. Springer.

[7] Billy Bob Brumley and Risto M. Hakala. 2009. Cache-Timing Template Attacks. In *Intl. Conf. on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*.

[8] Pablo Cañones, Boris Köpf, and Jan Reineke. 2017. Security analysis of cache replacement policies. In *International Conference on Principles of Security and Trust*.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[10] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. 2012. Side-channel Vulnerability Factor: a Metric for Measuring Information Leakage. In *ACM/IEEE Intl. Symp. on Computer Architecture*.

[11] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security Symposium*.

[12] Leonid Domnitser, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2010. 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*.

[13] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable Caches: Low-complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization* (2012).

[14] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2017. Verification of a practical hardware security architecture through static information flow analysis. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[15] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference*. Springer, 444–461.

[16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.

[17] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment*.

[18] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*.

[19] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games — Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy*.

[20] Zecheng He and Ruby B Lee. 2017. How secure is your cache against side-channel attacks?. In *IEEE/ACM International Symposium on Microarchitecture*.

[21] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing — and Its Application to AES. In *IEEE Symposium on Security and Privacy*.

[22] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*.

[23] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203* (2018).

[24] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in cryptology-CRYPTO 99*. Springer, 789–789.

[25] Boris Köpf and David Basin. 2007. An information-theoretic model for adaptive side-channel attacks. In *ACM Conf. on Computer and Comms. Security*.

[26] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic quantification of cache side-channels. In *Intl. Conference on Computer Aided Verification*.

[27] Yann Le Cun, LD Jackel, B Boser, JS Denker, HP Graf, I Guyon, D Henderson, RE Howard, and W Hubbard. 1989. Handwritten Digit Recognition: Applications of Neural Network Chips and Automatic Learning. *IEEE Communications Magazine* 27, 11 (1989), 41–46.

[28] Peng Li, Debin Gao, and Michael K. Reiter. 2014. StopWatch: A Cloud Architecture for Timing Channel Mitigation. *ACM Transactions on Information and System Security* (2014).

[29] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices.. In *USENIX Security Symposium*.

[30] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[31] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture*.

[32] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture*.

[33] Fangfei Liu, Hao Wu, Ken Mai, and Ruby B. Lee. 2016. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. *IEEE Micro* 36, 5 (2016).

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*.

[35] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. 2015. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *ACM Conference on Computer and Communications*

Security.

[36] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *RSA Conference on Topics in Cryptology*. 1–20.

[37] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan*.

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

[39] Jean-Jacques Quisquater and David Samyde. 2001. Electromagnetic Analysis (ema): Measures and Counter-measures for Smart Cards. *Smart Card Programming and Security* (2001), 200–210.

[40] C. Rebeiro and D. Mukhopadhyay. 2012. Boosting Profiled Cache Timing Attacks With A Priori Analysis. *IEEE Trans. on Information Forensics and Security* (2012).

[41] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The annals of mathematical statistics* (1951), 400–407.

[42] Frank Rosenblatt. 1958. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological review* 65, 6 (1958), 386.

[43] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning Representations by Back-propagating Errors. *nature* 323, 6088 (1986), 533.

[44] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. 2011. Limiting Cache-based Side-channel in Multi-tenant Cloud using Dynamic Page Coloring. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*.

[45] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. 2009. 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*.

[46] Josef Svenningsson and David Sands. 2010. Specification and verification of side channel declassification. In *Intl. Conf. on Formal Aspects in Security and Trust*.

[47] Kris Tiri, Onur Acıiçmez, Michael Neve, and Flemming Andersen. 2007. An analytical model for time-driven cache attacks. In *International Workshop on Fast Software Encryption*.

[48] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. 2011. Eliminating Fine Grained Timers in Xen. In *ACM Workshop on Cloud Computing Security*.

[49] Zhenghong Wang and Ruby.B. Lee. 2008. A Novel Cache Architecture with Enhanced Performance and Security. In *IEEE/ACM International Symposium on Microarchitecture*.

[50] Zhenghong Wang and Ruby B. Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *Annual Computer Security Applications Conference*.

[51] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *ACM International Symposium on Computer Architecture*.

[52] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*.

[53] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[54] Tianwei Zhang, Si Chen, Fangfei Liu, and Ruby B. Lee. 2013. Side Channel Vulnerability Metrics: the Promise and the Pitfalls. In *Hardware and Architectural Support for Security and Privacy*.

[55] Tianwei Zhang and Ruby B. Lee. 2014. New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels. In *Annual Computer Security Applications Conference*.

[56] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *ACM Conference on Computer and Communications Security*.

[57] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM Conference on Computer and Communications Security*.

[58] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *ACM Conference on Computer and Communications Security*.

[59] Yinqian Zhang and Michael K. Reiter. 2013. DüPpel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *ACM Conference on Computer and Communications Security*.

[60] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. 2016. A Software Approach to Defeating Side Channels in Last-level Caches. In *ACM Conference on Computer and Communications Security*.