A Formal Methodology for Verifying Side-Channel Vulnerabilities in Cache Architectures

Ke Jiang[1(✉)], Tianwei Zhang[1(✉)], David Sanán[2], Yongwang Zhao[3,4], and Yang Liu[1]

[1] School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore
ke006@e.ntu.edu.sg, tianwei.zhang@ntu.edu.sg

[2] Information and Communication Technologies, Singapore Institute of Technology, Singapore, Singapore

[3] ZJU-Hangzhou Global Scientific and Technological Innovation Center, Zhejiang University, Hangzhou, China

[4] School of Cyber Science and Technology, College of Computer Science and Technology, Zhejiang University, Hangzhou, China

Abstract. Security-aware CPU caches have been designed to mitigate side-channel attacks and prevent information leakage. How to validate the effectiveness of these designs remains an unsolved problem. Prior works assess the security of architectures empirically without a formal guarantee, making the evaluation results less convincing. In this paper, we propose a comprehensive methodology based on formal methods for security verification of cache architectures. Specifically, we design an entropy-based noninterference reasoning framework with two unwinding conditions to assess the information leakage of the cache designs. The reasoning framework quantifies the dependency relationships by the mutual information between the distributions of input and output of side channels. Given a cache design, we formalize its behavior specification along with the cache layouts into an abstract state machine, to instantiate the parameterized reasoning framework that discloses any potential vulnerabilities. We use our methodology to assess eight state-of-the-art cache architectures to demonstrate reliability as well as flexibility.

Keywords: Cache designs · Side-channel attacks · Security verification

1 Introduction

Micro-architectural side-channel attacks have incurred serious threats to computer security over the past decades [19]. These side-channel attacks mainly exploit the timing observations from hardware components (e.g., CPU cache [21,23], Translation Look-aside Buffer (TLB) [3,10]) to infer confidential information. The essence of these attacks is the interference [9] from the memory accesses between different programs or even inside one program. Such interference leaves regular footprints on certain hardware components, which can be

A. Riesco and M. Zhang (Eds.): ICFEM 2022, LNCS 13478, pp. 190–208, 2022.
https://doi.org/10.1007/978-3-031-17244-1_12

captured by an adversary to recover confidential information about the victim program. Past works have demonstrated successful attacks to steal cryptographic keys (symmetric ciphers [2], asymmetric ciphers [18,35,38], signature algorithms [1,25,34], post-quantum ciphers [11]), keystrokes [29], visited websites [26], and system configurations [14]. In this paper, we focus on cache-based side channels.

To mitigate cache side-channel attacks, a variety of defense solutions have been proposed. One promising direction is to design security-aware hardware components to reduce or prevent side-channel information leakage. These designs mainly follow two kinds of strategies. The partitioning-based solutions [31] physically partition the shared cache components into multiple zones for different domain applications to achieve strong isolation. The randomization-based solutions [24,31–33] obfuscate the adversary's observations by randomizing the cache states. These architectures exhibit great generalization and efficiency in protecting the programs running atop them. Although these architectures have been thoroughly considered and evaluated by researchers during the design phase, *it is still important to check whether there are any potential security vulnerabilities in these sophisticated cache systems before fabricating the actual chips.*

Over the past years, various methods have been proposed to evaluate cache side-channel vulnerabilities in hardware components. Unfortunately, they suffer from certain limitations, making it hard to apply them for practical and comprehensive verification. Specifically: **(1)** Some works [24,31–33] simulate the mechanisms of the newly designed caches against different types of side-channel attacks and empirically evaluated their effectiveness. Due to the lack of formal verification, they are not comprehensive, and can possibly miss some side-channel vulnerabilities. It also takes a lot of time to perform the cycle-accurate simulations in order to obtain convincing evaluation results. **(2)** A couple of approaches [7,8,30] abstractly describe the cache behaviors and define the execution paths that are treated as suspicious behaviors under a side-channel attack. Thereafter, they exhaustively search whether these suspicious behaviors are hidden in the cache behavior combinations. However, the modeling process is not formally guaranteed. Besides, the analysis is based on the exhaustive exploration of the execution traces, which can easily suffer from the combinatorial explosion issue. **(3)** Another challenge in verifying cache architectures is their probabilistic behaviors. To handle this issue, some works introduce methods based on statistics and entropy for security analysis. Zhang et al. [37] formally construct a cache state transition simulation through model checking techniques to get stable probability matrices within finite steps. To quantify information leakage, they calculate the mutual information between the input distribution and observable outputs. He et al. [13] establish a probabilistic information flow graph to model the interaction between the victim and attacker programs in the CPU cache. They define the concept of security-critical paths as the union of an attacker's observation and a victim's information flow. Equal probability of each node throughout the security-critical paths means the attacker cannot distinguish the victim's cache-accessed information. These methods face the balance issue between probability accuracy and state explosion. Besides, manual analysis and associating the probability to hardware behaviors can lead to incomprehensive

conclusions. **(4)** New hardware description languages (HDL) were introduced to design secure hardware circuits [6,16,36] with formal proof. These solutions are not comprehensive for side-channel analysis: they can only be applied to the partitioning-based caches while failing to evaluate the randomization-based designs with stochastic behaviors. Besides, they are not user-friendly and need manual work for attaching security labels and defining security policies.

To overcome the limitations of assessing the security of cache designs, this paper introduces a novel methodology based on formal methods by theorem proving to comprehensively verify the security of cache architectures against side-channel attacks. First, we formalize the specifications of cache designs in an event-state machine way. We offer functional correctness proofs to guarantee the consistency between the specifications and designs, which is ignored in prior works. Second, we design a noninterference reasoning framework to verify the side-channel vulnerability resident in the cache specifications. It adopts the concept of entropy [4,5] as the theoretical basis to assess the information leakage. We propose two unwinding conditions to unify and evaluate different types of secure caches (e.g., partitioning-based, randomization-based), making our solution comprehensive. Third, we implement our framework in Isabelle/HOL [20], and adopt it to verify eight state-of-the-art cache designs. In summary, we make the following contributions:

- We implement a noninterference reasoning framework based on information entropy that unifies both the deterministic and non-deterministic event models. We define nonleakage as security property by mutual information and derive two general unwinding conditions. We design interfaces for this framework to offer verification services.
- We formally specify each cache design in an event-state machine way on top of general set-associative cache layouts, forming a complete cache specification. We prove the cache specification is an instantiation of the reasoning framework, hence can be efficiently verified security properties.
- We evaluate our entropy-based noninterference reasoning framework on eight state-of-the-art cache designs. The verification practice shows that our methodology possesses high theoretical reliability and flexibility.

We present the background about cache side-channel attacks and mutual information in Sect. 2. We give the threat model and briefly describe the methodology in Sect. 3. The design of reasoning framework is shown in Sect. 4. Section 5 presents a case-study, and Sect. 6 analyzes the verification results of eight state-of-the-art cache architectures. Section 7 concludes this paper.

2 Background

2.1 Cache Side-Channel Attacks

Cache Hierarchy. Most CPU caches are organized in a n-way set-associative way. A n-way set-associative cache can be treated as a two-dimensional data array. Each row is called a *cache set*, which is further divided into n *cache lines*. Each memory block is mapped to one cache set indexed by its memory

address. This block can be stored in any cache lines in this set, determined by a replacement policy. When a CPU core wants to access a memory block, if it resides in the cache, the CPU can directly obtain it, resulting in a cache hit with a fast access speed. The CPU has to fetch the data from the main memory to the cache, otherwise. This results in a cache miss with a much slower access speed. Particularly, a cache with only one way in each set (i.e., $n = 1$) is a direct-mapped cache, while a cache with only one set is called fully-associative.

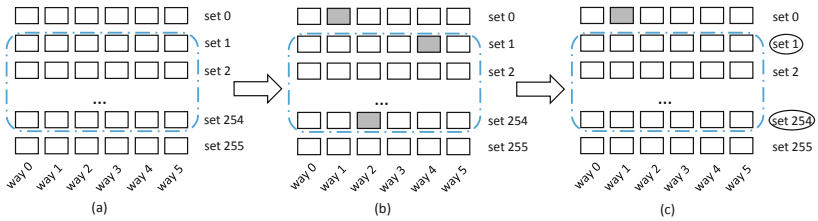

Fig. 1. Side-channel attack scheme. Sub-figure (a) represents the preparation phase, (b) the waiting phase, and (c) the observation phase.

Cache Attacks. The first cache attacks deduce cryptographic secrets by observing the whole execution time [2,15,22,27]. In recent days, cache side-channel attack techniques narrow down to a smaller granularity. The timing difference between a cache hit and a cache miss can reveal information about the program's access traces. A cache side-channel attack typically involves three steps (see Fig. 1). (1) Preparation: the adversary manipulates the states of certain cache lines with its own address space [28]. For instance, PRIME-PROBE attack [21] fills up the entire critical cache sets, while a FLUSH-RELOAD attack [35] and a FLUSH-FLUSH attack [12] evict certain cache lines through the *clflush* instruction. The area controlled by the adversary is shared with the victim. And, the adversary has the knowledge that the victim's access pattern in this area is related to its confidential information. (2) Waiting: the adversary does nothing until the victim finishes several execution circles. The victim may load its data blocks into the cache and replace the cache lines occupied by the adversary. (3) Observation: the adversary collects the footprints left by the victim program. For example, the PRIME-PROBE attack re-accesses the critical cache set to check if certain blocks were evicted by the victim. The FLUSH-RELOAD attack reloads the target cache lines to determine if it has been touched by the victim. The FLUSH-FLUSH attack re-flushes the target cache lines to check whether data is loaded into these lines by the victim. The victim's cache access pattern is thus leaked to the attacker.

2.2 Mutual Information

Intuitively, the concept of noninterference tells that one domain (denoted as victims) does not affect the observation of another domain (denoted as adversaries). The concept is consistent with the cache side-channel attack schemes because an

adversary can deduce the victim's memory access patterns that are associated with secrets when its observation depends on the victim's behaviors. Furthermore, from the perspective of the probability distribution, information leakage of the side-channel is equivalent to the existence of a dependency relationship between the input (victim's behaviors) and output (adversary's observation). And, this kind of dependency relationship can be calculated by mutual information. This is the motivation of this work where we interpret the cache side-channel attack schemes by noninterference and measure the information flow of this noninterference through mutual information of Shannon Theory [4,5].

We denote a victim's behaviors as the uncertain information that an attacker wishes to explore by side-channel attacks. This information is viewed as input X and has probability distribution $\mathcal{X}$. First, entropy defines the uncertainty of the information itself, of $H(X) = -\sum_{x\in\mathcal{X}} p(x)log_2p(x)$. Second, conditional entropy measures the uncertainty about X when the attacker has the knowledge of output Y. It is defined as $H(X|Y) = -\sum_{y\in\mathcal{Y}} p(y)\sum_{x\in\mathcal{X}} p(x|y)log_2p(x|y)$. Lastly, mutual information between X and Y measures the information that an adversary can learn about X if he gains the knowledge through output Y, defined as $I(X;Y) = H(X) - H(X|Y)$. One property of mutual information is that it is symmetry: $I(X;Y) = I(Y;X)$. It can be calculated through a joint probability matrix, as shown in Eq. 1.

$$I(X;Y) = \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)log_2\frac{p(x,y)}{p(x)p(y)} \tag{1}$$

$$= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x)p(y|x)log_2\frac{p(y|x)}{p(y)} \tag{2}$$

2.3 Isabelle/HOL

Isabelle/HOL [20] is a higher-order logic theorem prover. It offers common types (e.g., naturals (*nat*), integers (*int*) and booleans (*bool*)). The keyword *datatype* is used to define an inductive data type. Composed data types include tuple, record, list, and set. Projection functions *fst* and *snd* return elements t_1 and t_2 of a tuple $(t_1 \times t_2)$. Isabelle/HOL offers record type to include multiple elements of different data types. Assignment symbol = is used to initialize the contents of a record, while := is used to update it. Lists are defined by an empty list denoted as [], and a concatenation constructor represented as #. The ith component of a list xs is accessed by $xs!i$. The cardinality of a set s (i.e., $|s|$) is denoted as *card* s, returning zero when set s is infinite. Isabelle/HOL provides *definition* command to specify a non-recursive function, while *primrec* command for primitive recursions.

Isabelle/HOL supports parametric theories with the keyword *locale*. A *locale* includes a series of parameter declarations (with keyword *fixes*) and assumptions (with keyword *assumes*). Isabelle/HOL users can instantiate a locale through *interpretation* command, where concrete data is assigned to parameters and declarations are added to the current context. We construct a parametric non-interference reasoning framework through *locale* command and instantiate it in different cache architecture scenarios through *interpretation* command.

3 Methodology Overview

3.1 Threat Model

We consider the cache architectures to be verified are involved in the following threat model. The victim and the attacker share the same cache environment and a cross-core/VM attack allows the attacker and the victim to execute in parallel on different cores/VMs. The attacker cannot directly observe the memory content from the CPU, probing cache states to check whether the victim's data is resident in the cache indirectly instead. This model captures most cache side-channel attacks in the literature. For example, EVICT-TIME attack [21] measures the latency of victim's program, PRIME-PROBE attack [18,21,23], FLUSH-RELOAD attack [35] and FLUSH-FLUSH attack [12] measure the latency of attacker's program.

We also mention that the attacker accurately monitors both cache set and cache line granularities. This is because modern OS adopts the page sharing technique that removes the duplication of shared libraries, enabling probing the shared libraries narrow to a cache line.

3.2 Architecture

The workflow of our proposed methodology is shown in Fig. 2. It includes two components. (1) A reasoning framework is designed to quantify the information leakage of the target system. The essence of the framework is to interpret the noninterference property through mutual information. (2) A complete cache specification includes the cache behavior formalization and the general cache layouts. The cache behavior is described as an event-state transition. Its formalization is first proved to meet the consistency with its design. The cache specification instantiates the interface layer offered by the reasoning framework. Therefore, for verifying whether a cache specification satisfies the security properties, we only need to verify whether it satisfies two unwinding conditions. Violations of both conditions indicate the cache design is vulnerable to side-channel attacks. In this work, we mainly focus on the fundamentals of information leakage, while skipping the analysis of adversarial strategies for extracting the secrets from the footprints of the victim program. As shown in Fig. 2, the reasoning framework contains the following components.

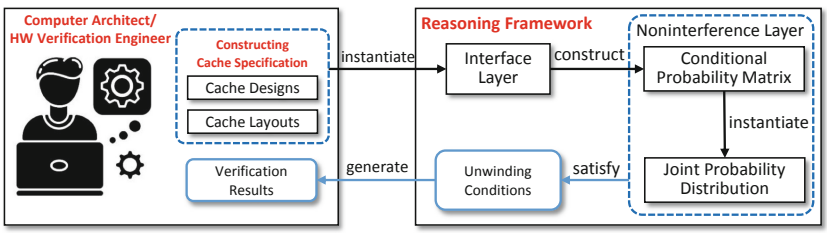

Fig. 2. Workflow of our proposed approach.

Interface Layer: this layer is used to connect the given cache specification to the noninterference layer. The interface layer offers an event-state transition function and output function to parse cache behaviors into probabilistic representations.

Noninterference Layer: this is the core of our reasoning framework. It introduces a parameterized joint probability distribution between the victim's information X and attacker's observation Y. We calculate the mutual information $I(X;Y)$ as information flow security property to quantify how much information the attacker can learn about X from Y. The joint probability distribution can be constructed by multiplying a series of probabilistic inputs and a conditional probability matrix. The conditional probability matrix models the relationships between the input and the output of cache designs.

Unwinding Conditions: this defines the conditions to satisfy information flow security property, meaning to make the mutual information zero according to Eq. 2. We deduce two unwinding conditions as shown in Eq. 3, when we stipulate that each input probability is greater than zero. The first condition $C1$ indicates that the attacker learns nothing when there is no observation. The second condition $C2$ shows the attacker's observation is constant and independent of the victim's behaviors.

$$\begin{aligned} C1 &: \forall x\ y.\ p(y|x) = 0 \longrightarrow I(X;Y) = 0 \\ C2 &: \forall x\ y.\ p(y|x) = p(y) \longrightarrow I(X;Y) = 0 \end{aligned} \tag{3}$$

4 Design of Reasoning Framework

In this section, we provide more details about our reasoning framework.

4.1 Interface Layer—An Abstract State Machine

A cache specification implements a state machine through instantiating the interface layer. We construct the interface layer for the purpose of re-usability because verification of any cache architecture only requires instantiating the interface functions.

First, we model the input distribution space as $\mathbb{P}(\mathcal{I} \times \mathcal{P})$, which is the powerset of type $\mathcal{I} \times \mathcal{P}$. Label $\mathcal{I}$ describes the input content and $\mathcal{P}$ is of real type describing probabilities. We further stipulate any valid input distribution is neither empty nor infinite[1]. We omit input elements with zero-probability because they will not result in any outputs. We also guide that all inputs are different and the sum of the probabilities of all inputs equals one. We use the operator . to denote an attribute of a input, e.g., $x.i$ and $x.p$ represent the content and probability of input x, respectively.

Definition 1 (Valid Input Distribution Sets).

$$\begin{aligned} makeInput \triangleq \{\mathcal{X}.\ \mid \mathcal{X} \mid > 0 \wedge (\forall m\ n \in \mathcal{X}.\ m \neq n \longrightarrow m.i \neq n.i) \wedge \\ (\forall d \in \mathcal{X}.\ d.p > 0) \wedge \sum_{d \in \mathcal{X}} d.p = 1\} \end{aligned}$$

[1] We follow the Isabelle/HOL definition where the cardinality of infinite sets is zero.

Next, we formalize the event-state transition function as ψ, which describes a non-deterministic event model. It is a single step execution triggered by the event label and the input, of type $\mathcal{E} \times \mathcal{X} \to \mathbb{P}(\mathcal{S} \times \mathcal{S})$. Label $\mathcal{S}$ represents the state space and $\mathcal{E}$ is the set of event labels. We use $\psi(e, x)/s$ to represent that all event-state transitions happen when we execute the event e on the state s with input x.

Definition 2 (Event-state Transition Function from State s).

$$\psi(e, x)/s \triangleq \{t.\ t \in \psi(e, x) \wedge (\exists s'.\ t = (s, s'))\}$$

Then, we define an abstract output function that extracts the output from each transition tuple (s, s'), of type $\varpi : (\mathcal{S} \times \mathcal{S}) \to \mathcal{O}$. Label $\mathcal{O}$ describes the output content. With these functions, we construct an abstract state machine in the interface layer.

Definition 3. *The interface layer implements an abstract state machine $\mathcal{M}$ as tuple $\langle \mathcal{S}, \mathcal{E}, \mathcal{X}, \mathcal{O}, \psi, \varpi \rangle$, where $\mathcal{S}$ is the state space, $\mathcal{E}$ is the set of event labels, $\mathcal{X}$ and $\mathcal{O}$ are the valid input distribution and output content respectively, ψ is the event-state transition function, and ϖ is the output function.*

4.2 Noninterference Layer

The concept of noninterference indicates that the behaviors of one domain do not affect the observation of another domain [9]. In our reasoning framework, these two domains correspond to the victim's input and the attacker's observation in the side channel. To describe such a side-channel mechanism, we construct a joint probability distribution through the functions from the interface layer step by step. We quantify the effect of the interaction between the victim and attacker by calculating mutual information from the joint probability distribution.

A joint probability can be written as $P(X)P(Y|X)$. Therefore, to instantiate a joint probability distribution is to construct the input distribution $P(X)$ and a conditional probability matrix $P(Y|X)$. The input distribution can be directly inherited from the interface layer. For example, it is any set that satisfies $\mathcal{X} \in makeInput$.

To construct a conditional probability matrix, we first introduce a conditional probability transition function Cpt. It first applies output function to each state transition tuple to get all possible outputs, shown as $\mathcal{O} = \{d.\ \exists t \in \psi(e, x)/s.\ d = \varpi(t)\}$. Then, for each output $o \in \mathcal{O}$, it counts all the transitions that produce the output o to get the probability, which is the proportion of these transitions in the total transitions. The definition of Cpt is as follows. The result of this function is the output distribution $\mathcal{Y}$, of type $\mathbb{P}(\mathcal{O} \times \mathcal{P})$.

Definition 4 (Conditional Probability Transition Function).

$$Cpt(e,x)/s \triangleq \{y.\, \exists o \in \mathcal{O}.\ \mathcal{T}_{sub} = \{t.\ t \in \psi(e,x)/s \wedge \varpi(t) = o\} \wedge y.o = o \wedge y.p = \frac{\mid \mathcal{T}_{sub} \mid}{\mid \psi(e,x)/s \mid}\} \quad where\, \mathcal{O} = \{d.\ \exists t \in \psi(e,x)/s.\ d = \varpi(t)\}$$

The function Cpt only takes one input while the valid input distribution $\mathcal{X}$ contains limited input contents. Therefore, the next step is to apply the function Cpt to each input in $\mathcal{X}$. The result of this process is a conditional probability matrix $\mathcal{W}$. Each row of the matrix ($w[y_1|x_i]$, $w[y_2|x_i]$ ... $w[y_{|Y|}|x_i]$) can be viewed as the representation of the conditional probability distribution of output $y_1,\ y_2 \ldots y_{|Y|}$ under the input x_i.

Now it is time to build the joint probability distribution. The following function $makeJoint$ matches each input x that belongs to the input distribution $\mathcal{X}$ with any conditional probability y that is part of $Cpt(e,x)/s$. Then the joint probability is the product of the corresponding probabilities of these two elements. Joint distribution $\mathcal{J}$ is defined as $\mathbb{P}((\mathcal{I} \times \mathcal{O}) \times \mathcal{P})$.

Definition 5 (Joint Probability Distribution).

$$makeJoint \triangleq \{j.\, \exists x \in \mathcal{X}.\ \exists y \in Cpt(e,x)/s.\ j.i = x.i \wedge j.o = y.o\ \wedge j.p = x.p * y.p\}$$

The computation of mutual information from Eq. 1 requires two marginal probability distributions. We take the marginal probability of the input x as an example: we first collect the subset $\mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.i = x.i\}$ that takes all elements whose input dimension is equal $x.i$ from the joint probability distribution. Then the marginal probability is the sum of the probabilities of all such elements. Its definition is shown below. We omit the definition of $margOutput$ due to the space limit.

Definition 6 (Input Marginal Probability Distribution).

$$margInput \triangleq \{mi.\, \exists x \in \mathcal{X}.\ \mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.i = x.i\} \wedge mi.i = x.i \wedge mi.p = \sum_{d \in \mathcal{J}_{sub}} d.p\}$$

Now we give the definition of mutual information based on Eq. 1. Function $mutualInfo$ takes each element $j \in makeJoint$ from the joint probability distribution, and then calculates the marginal probabilities of the input (mi) and output (mo) respectively. Afterwards, the value of the mutual information is the accumulation of $j.p * log_2 \frac{j.p}{mi.p * mo.p}$, when iterating the element j.

Definition 7 (Mutual Information).

$$mutualInfo \triangleq \sum_{j \in makeJoint} j.p * log_2 \frac{j.p}{mi.p * mo.p}$$

4.3 Unwinding Conditions

With mutual information calculated above, we assess the information leakage of noninterference by the following definition.

Definition 8 (Information Leakage).

$$nonleakage \triangleq \forall e\ \mathcal{X}\ s.\ mutualInfo = 0$$

According to Eq. 3, two unwinding conditions imply that the mutual information equals zero. We give the definitions of these two unwinding conditions and prove the implication relationships.

Theorem 1 (Condition 1: No Observation).

$$\forall e\ s.\ \forall x \in \mathcal{X}. \forall y \in Cpt(e,x)/s. \\ y.p = 0 \longrightarrow mutualInfo = 0$$

Proof. When the conditional probability $y \in Cpt(e,x)/s$ equals zero, the corresponding joint probability $j.p = x.p * y.p$ also equals zero. Then unfolding the definition of $mutualInfo$ and substituting $j.p$ as 0, the accumulated result $0 * log_2 \frac{0}{mi.p*mo.p}$ is zero. In the end, the mutual information is zero.

Theorem 1 gives the advice that if the attacker cannot observe anything from the footprints released by the victim, then the cache dose not leak any information. This condition can be used in some partitioning-based designs [31].

Theorem 2 (Condition 2: Constant Observation).

$$\forall e\ s.\ \forall x \in \mathcal{X}.\ \forall y \in Cpt(e,x)/s. \\ y.p = \sum_{d \in \mathcal{J}_{sub}} d.p \longrightarrow mutualInfo = 0 \\ where\ \mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.o = y.o\}$$

Proof. For any joint probability $j \in makeJoint$, its corresponding marginal probability of the input $mi.p$ equals its input probability $x.p$ because the sum of the probabilities of all elements in $Cpt(e,x)/s$ equals one. Also, the joint probability $j.p$ can be calculated by $x.p * y.p$. We have $y.p = \sum_{d \in \mathcal{J}_{sub}} d.p$, where $\mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.o = y.o\}$ according to the condition 2 above. The accumulated result in the definition of $mutualInfo$, $j.p * log_2 \frac{j.p}{mi.p*mo.p}$ of Eq. 1, is then folded and substituted as $x.p * \sum_{d \in \mathcal{J}_{sub}} d.p * log_2 \frac{x.p * \sum_{d \in \mathcal{J}_{sub}} d.p}{x.p * \sum_{d \in \mathcal{J}_{sub}} d.p}$. Its value equals zero, leading the mutual information to be zero as well.

Theorem 2 describes a scenario where the conditional probability distribution is constant for any input. Therefore, the footprints caused by any input are the same. Note that Theorem 2 only requires the values of each column in the matrix $\mathcal{W}$ to be the same. When this condition is applied into cache designs, we can find

that some randomization-based strategies further manipulate that the values of $w[y_1|x_i],\ w[y_2|x_i] \ldots w[y_{|Y|}|x_i]$ are in the same probability, which is one special case of the above condition.

In the end, either of these two unwinding conditions can imply no information leakage, as shown in the following theorem.

Theorem 3 (Unwinding Conditions Reasoning).

$$(condition1 \vee condition2) \Longrightarrow nonleakage$$

5 Application of Our Methodology: A Case-study

In this section, we demonstrate how to verify an existing randomization-based cache design, i.e., Random Permutation (RP) cache [31] with our methodology.

5.1 The General Cache Layouts Specification

We start with the specification of the general cache layouts. A cache line is the smallest unit among cache layouts, which is defined as a **record** $ca\_line = ca\_set,\ ca\_way,\ ca\_tag,\ valid,\ lock,\ owned$. The first three fields directly represent the cache index, cache way and cache tag. The following fields denote whether the cache line is used, whether its content is protected, and which process is occupying it. We define the cache structure and the specification it needs to satisfy as follows: the parameterized cache layouts are constructed by a list whose length is **M**, where each element of the list represents a cache set with **W** cache lines (i.e., W-ways). In a cache set, the *ca_set* identifier of each cache line equals its cache set index, and all cache lines in one cache set have different *ca_way*.

Definition 9 (The Cache Structure).

$$\begin{aligned}
&Cache :: \text{“}(ca_line\ set)\ list\text{”}\ and\ it\ satisfies : |\ Cache\ | = M\ \wedge \\
&(\forall l < M.\ \ |\ Cache\ !\ l\ | = W) \wedge (\forall l < M.\ \forall e \in (Cache\ !\ l).\ e.ca_set = l)\ \wedge \\
&(\forall l < M.\ \forall e_i\ e_j.\ e_i \in (Cache\ !\ l) \wedge e_j \in (Cache\ !\ l). \\
&\qquad e_i \neq e_j \longrightarrow e_i.ca_way \neq e_j.ca_way)
\end{aligned}$$

With the cache layouts definition, we formalize the memory request that acts as the input from the victim and is performed by the corresponding cache design on the cache layouts. A memory request is denoted as a **record** $mem\_req = tagbits,\ setbits,\ protected,\ process$. Label *tagbits* is used to compare the tag field of a cache line, and *setbits* is sent to the cache mapping to get the actual cache index. Label *protected* denotes whether the memory data is protected by its owner, and *process* represents the owner of this memory block in two values: **H** and **L** denote the confidential and non-confidential processes respectively. Last, we design the system state that includes the cache structure and the mapping structure from a memory request to the cache set index. The structure is formally defined as **record** $state = Cache,\ Mapping$.

5.2 The RP Cache Specification

For the RP cache, the workflow of handling memory requests is shown in Fig. 3. RP cache utilizes three strategies to randomize the observation of the adversary. (1) When there is an external cache miss (Column 1: the mapped cache line does not belong to the current process), RP cache randomly chooses a cache set and selects one cache line according to the replacement policy in this set to replace the request memory. (2) When there is an internal miss (Column 2: the mapped cache line belongs to the current process but has a different protection flag), RP cache randomly chooses a cache set and selects one cache line according to the replacement policy in this set to evict it without caching. These two strategies will result in non-deterministic cache state transitions. (3) For the external cache miss (Column 1), RP cache also dynamically changes the memory-to-cache mapping, so even if the attacker can deduce the mapping for one read operation, it would fail for the next time.

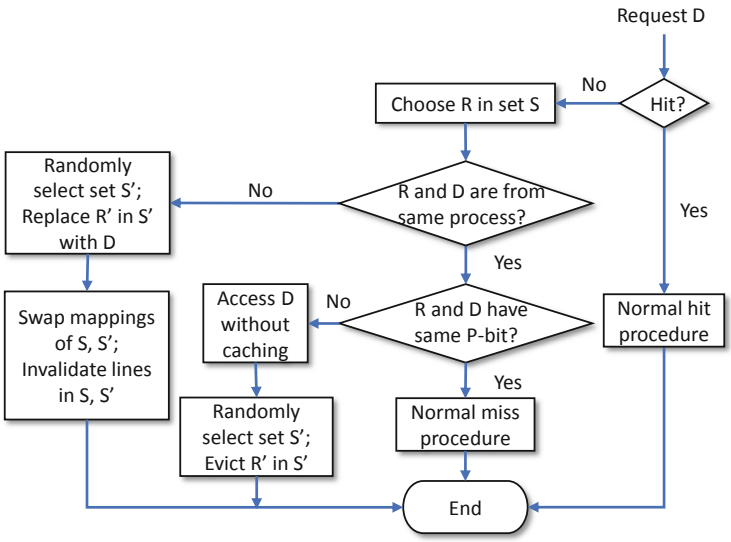

Fig. 3. Random permutation cache

The specification of RP cache is defined in a way of the event-state transition function and is omitted here. We give a lemma to prove the correctness of cache layout when a RP cache read is issued and a miss happens. It indicates that there will be multiple state transitions when a cache miss happens, showing non-deterministic execution. For each transition, there will be one cache line from a random cache set updated, while other cache sets remain the same. Here, l_{mr} refers to the original mapped cache set index.

Lemma 1 (Correctness of RP Cache Read).

$$[\![\forall e \in s.Cache\ !\ l_{mr}.\ e.ca\_tag \neq mr.tagbits]\!] \Longrightarrow$$
$$[\![\forall s' \in rp\_read\ mr\ s.\ \exists! l.\ (\forall l' \neq l.\ s'.Cache\ !\ l' = s.Cache\ !\ l') \wedge$$
$$|\ (s'.Cache\ !\ l) \cap (s.Cache\ !\ l)\ | = |\ s.Cache\ !\ l\ | - 1]\!]$$

5.3 Security Verification of RP Cache

With the instantiation of the event-state transition function (i.e., the RP cache specification), we instantiate the remaining two functions to complete the instantiation of the interface layer of the reasoning framework.

Due to direct inheritance, the memory request distribution issued from the victim is regarded as $\mathcal{X}$, of concrete type $\mathbb{P}(mem\_req \times \mathcal{P})$. Next, we instantiate the observation function ϖ. According to the correctness proofs, only one cache set is updated inside those state transitions for each cache miss. Therefore, the attacker can regard the state transition with the same updated cache set as the same observation when re-accessing the cache. For convenience, we use the cache set index to represent the observable state transitions.

Now, we construct the attack-simulated cache layout specification that statically demonstrates how the attacker manipulates the cache layout with its data. It describes the circumstances of the preparation phase shown in the leftmost picture of Fig. 1. A cache line e can be differentiated through identifier $e.owned = \mathbf{H}$ and $e.owned = \mathbf{L}$, denoting e is occupied by a confidential or non-confidential process. An attacker use a series of memory accesses without cache collision to fill part of the cache. The specification of manipulated cache below indicates that the attacker's accesses to each memory address in the m_s will result in a cache hit. Thereafter, the victim's access to these manipulated cache areas will change their ownership, resulting in observation to the attacker if re-accessing this memory space.

Definition 10 (The Attack-simulated Cache Layout).

$$m\_s :: \text{“}mem\_req\ set\text{”}\ and\ it\ satisfies :$$
$$\forall m \in m\_s.\ (m.process = L) \wedge (\exists! e \in Cache\ !\ (Mapping \rightharpoonup m.setbits).$$
$$e.ca\_tag = m.tagbits \wedge e.owned = L)$$

As for the replacement policy in a cache set, we follow the practice in [18] where they consider the age replacement (e.g., LRU replacement or FIFO replacement) and stipulate that the adversary re-accesses a cache set in a reversed order. In such a way, the victim evicts the oldest cache line at the adversary's waiting state, shown in the middle picture of Fig. 1. If the adversary probes the cache set in his original order, then the second-oldest (same for the followings) cache line is evicted, leading to a miss on every probe. In contrast, if the attacker probes the cache set in a reversed order, the cache line evicted by the victim can be precisely probed without causing a miss on every probe. We leave the random replacement policy as future work.

With all the preparations above, we turn to the security verification of RP cache read operation. It breaks the first unwinding condition but preserves the second one, which means it produces constant observations to the attacker regardless of the victim's input. We prove the theorem of RP cache read as follows.

Theorem 4 (RP Cache Read Produces Constant Observation).

$$\forall x \in \mathcal{X}.\, Cpt(rp\_read, x)/s = \{y.\ \exists o \in \{0\ ..\ M-1\}.\ y = (o, \frac{1}{M})\} \wedge$$
$$(\forall y \in Cpt(rp\_read, x)/s.\ y.p = \sum_{d \in \mathcal{J}_{sub}} d.p)$$
$$where\ \mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.o = y.o\}$$

Proof. With the attack specification, any input that causes an external or internal cache miss requires the RP cache to select one cache line according to replacement policy from each set for replacement or eviction. Therefore, the range of the observation is the whole cache set index under the extreme circumstance, where the attacker can control the whole cache. This is shown in the first line of the above theorem. Meanwhile, the probability of each cache set index equals $\frac{1}{M}$, where M is the length of the whole cache index. When applying this knowledge to the reasoning framework, we can prove that the RP cache read operation satisfies the second unwinding condition. The observation range narrows under the non-extreme circumstance, while all the observations have a uniform probability. Therefore, there is no information leakage through this process.

6 Security Verification Results and Analysis

We successfully verify eight state-of-the-art cache designs. We implement all the verification work in Isabelle/HOL[2]. Table 1 shows the verification results, as well as the implementation complexity (lines of codes) for each cache. We give the security analysis of eight state-of-the-art cache designs as follows.

Table 1. Verification results of cache designs.

Cache design	No-observation	Constant-observation	Leakage	LOC
Set-Associative Cache	×	×	*yes*	*490+*
Random Fill Cache	×	×	*yes*	*930+*
Partition Locked Cache	√	○	*no*	*380+*
Random Permutation Cache	×	√	*no*	*1490+*
NewCache	×	√	*no*	*1260+*
CEASE Cache	×	×	*yes*	*510+*
CEASER Cache	×	×	*yes*	*580+*
SCATTER Cache	×	×	*yes*	*500+*

[2] Interested readers can refer here for the reasoning framework and verification cases.

Set-Associative (SA) Cache. This conventional cache is well known to be vulnerable to side-channel attacks. Among the cache regions controlled by the attacker, any memory request from the victim can cause an observation with a conditional probability of 1 due to the deterministic execution. Therefore, the cache operation breaks the first condition. Also, a program owns multiple memory address mapped to different cache sets, resulting in different observations. Then, there exists a marginal output that is not equal to the conditional probability of 1, breaking the second condition. Hence, SA cache leaks side-channel information.

Random Fill (RF) Cache [17]**.** RF cache fills the cache line to be replaced with a random memory line from a neighborhood window of the request memory line. It can result in observations to the attacker, breaking the first condition. Although it randomly picks up a memory line among a stated window, it cannot promise to produce the same range of observations and of equal probability. In extreme cases, RF cache can degrade to a SA cache if the neighborhood window is small, making each memory line mapped to the same cache set. This property cannot imply the second condition. Therefore, there exists information leakage.

Partition Locked (PL) Cache [31]**.** PL cache with prefetching strategy adds a lock mechanism to the cache line. A replacement policy will work only when the cache line chosen to be replaced is unlocked or belongs to the same process. Therefore, an attacker has no chance to obtain any observations. This cache design is the only one that satisfies the first unwinding condition.

Random Permutation (RP) Cache [31]**.** Any memory request that causes an external or internal cache miss requires the RP cache to select one cache line from each set for replacement or eviction. Therefore, the range of the observation is the whole cache set index under the extreme circumstance, where the attacker can control the whole cache. Meanwhile, the probability of each cache set index equals $\frac{1}{M}$, where M is the length of the whole cache index. We can prove that the RP cache operation satisfies the second unwinding condition. The observation range narrows under the non-extreme circumstance, while all the observations have a uniform probability. Therefore, there is no information leakage through this progress.

NewCache [32]**.** For the protected memory requests, NewCache employs similar strategies as the RP cache. Therefore, it responses to an external or internal cache access by selecting one cache line from each cache set to replace the memory block or to cause an eviction deliberately. Applying similar verification proves the security of NewCache against side-channel attacks.

CEASE Cache [24]**.** The CEASE cache employs encryption over the physical address and uses the ciphertext to index the cache. However, the memory to cache-set mapping remains constant as long as the encryption key remains the same. Unfortunately, it degenerates to a set-associative cache, which means an adversary can observe a deterministic change for each cache miss. As a result, the CEASE cache may leak confidential information.

CEASER Cache [24]**.** CEASER cache is an advanced version of CEASE, which adopts dynamic remapping to periodically change the key and remap the lines based on the new key. In the phase when both the current and next keys exist, previous remapping and the victim's access can provide useful observations to the attacker. Therefore, it breaks the first unwinding condition. Although part of the observations is created by remapping of cache lines, the attacker can obtain a deterministic observation during each epoch, and thus there exists such a marginal output that is not equal to its corresponding conditional probability. Therefore, it can not satisfy the second unwinding condition, and information leakage exits.

SCATTER Cache [33]**.** The SCATTER cache employs the index derivation function that takes the secure domain identifier, the encryption key, and the memory request as inputs to form a nominal cache set (the cache line of each cache way comes from different cache sets). The mapping table is deterministic to a process as long as the encryption key is constant. Another characteristic is that the conditional probability of each output is uniform, while the ranges of these outputs are inconsistent. Therefore, it can not satisfy the second unwinding condition, leaving a trail of telltale information.

7 Conclusions and Future Work

In this paper, we propose a novel verification methodology to verify side-channel vulnerabilities resident in the cache designs. We construct an entropy-based non-interference reasoning framework with two unwinding conditions for the evaluation of side-channel threats. We use our methodology to successfully assess and evaluate the security of eight state-of-the-art cache solutions. The verification practice indicates that our verification framework offers strong accuracy and persuasion for the verification of cache side channels.

Although our methodology provides a strong guarantee for the security verification of cache designs from the perspective of formal methods, it still has drawbacks. Our reasoning framework asks a high-level requirement of professionalism of theory proving on users and cannot offer an automated derivation process. Therefore, in future work, we plan to automate the validation process and provide more refined cache models.

Acknowledgements. This work has been supported in part by Singapore National Research Foundation under its National Cybersecurity R&D Programme (NCR Award NRF2018 NCR-NCR009-0001), Singapore Ministry of Education (MOE) AcRF Tier 1 RS02/19, NTU Start-up grant, and the National Natural Science Foundation of China (NSFC) under the Grant No. 62132014 and by Key R&D Program of Zhejiang Province under the Grant No. 62132014.

References

1. Aranha, D.F., Novaes, F.R., Takahashi, A., Tibouchi, M., Yarom, Y.: LadderLeak: breaking ECDSA with less than one bit of nonce leakage. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 225–242 (2020)
2. Bernstein, D.J.: Cache-timing attacks on AES (2005)
3. Canella, C., et al.: Fallout: leaking data on meltdown-resistant CPUs. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 769–784 (2019)
4. Chatzikokolakis, K., Chothia, T., Guha, A.: Statistical measurement of information leakage. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 390–404. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_33
5. Cover, T.M.: Elements of Information Theory. Wiley, Hoboken (1999)
6. Deng, S., et al.: SecChisel framework for security verification of secure processor architectures. In: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, pp. 1–8 (2019)
7. Deng, S., Xiong, W., Szefer, J.: Cache timing side-channel vulnerability checking with computation tree logic. In: Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, pp. 1–8 (2018)
8. Deng, S., Xiong, W., Szefer, J.: Analysis of secure caches using a three-step model for timing-based attacks. J. Hardware Syst. Secur. **3**(4), 397–425 (2019)
9. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, pp. 11–11. IEEE (1982)
10. Gras, B., Razavi, K., Bos, H., Giuffrida, C.: Translation leak-aside buffer: defeating cache side-channel protections with TLB attacks. In: 27th USENIX Security Symposium (USENIX Security 2018), pp. 955–972 (2018)
11. Groot Bruinderink, L., Hülsing, A., Lange, T., Yarom, Y.: Flush, gauss, and reload – a cache attack on the BLISS lattice-based signature scheme. In: Gierlichs, B., Poschmann, A.Y. (eds.) CHES 2016. LNCS, vol. 9813, pp. 323–345. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53140-2_16
12. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+Flush: a fast and stealthy cache attack. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 279–299. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40667-1_14
13. He, Z., Lee, R.B.: How secure is your cache against side-channel attacks? In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 341–353 (2017)
14. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: 2013 IEEE Symposium on Security and Privacy, pp. 191–205. IEEE (2013)
15. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
16. Li, X., et al.: Sapper: a language for hardware-level security policy enforcement. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 97–112 (2014)
17. Liu, F., Lee, R.B.: Random fill cache architecture. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 203–215. IEEE (2014)

18. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE Symposium on Security and Privacy, pp. 605–622. IEEE (2015)
19. Lou, X., Zhang, T., Jiang, J., Zhang, Y.: A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. ACM Comput. Surv. (CSUR) **54**(6), 1–37 (2021)
20. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
21. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006). https://doi.org/10.1007/11605805_1
22. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive (2002)
23. Percival, C.: Cache missing for fun and profit (2005)
24. Qureshi, M.K.: CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 775–787. IEEE (2018)
25. Ryan, K.: Return of the hidden number problem. IACR Trans. Cryptogr. Hardware Embed. Syst. 146–168 (2019)
26. Shusterman, A., et al.: Robust website fingerprinting through the cache occupancy channel. In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 639–656 (2019)
27. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of DES implemented on computers with cache. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 62–76. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45238-6_6
28. Vila, P., Köpf, B., Morales, J.F.: Theory and practice of finding eviction sets. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 39–54. IEEE (2019)
29. Wang, D., et al.: Unveiling your keystrokes: a cache-based side-channel attack on graphics libraries. In: NDSS (2019)
30. Wang, L., Zhu, Z., Wang, Z., Meng, D.: Analyzing the security of the cache side channel defences with attack graphs. In: 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 50–55. IEEE (2020)
31. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. In: Proceedings of the 34th Annual International Symposium on Computer Architecture, pp. 494–505 (2007)
32. Wang, Z., Lee, R.B.: A novel cache architecture with enhanced performance and security. In: 2008 41st IEEE/ACM International Symposium on Microarchitecture, pp. 83–93. IEEE (2008)
33. Werner, M., Unterluggauer, T., Giner, L., Schwarz, M., Gruss, D., Mangard, S.: {ScatterCache}: thwarting cache attacks via cache set randomization. In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 675–692 (2019)
34. Yarom, Y., Benger, N.: Recovering OpenSSL ECDSA Nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive (2014)
35. Yarom, Y., Falkner, K.: FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: 23rd {USENIX} Security Symposium ({USENIX} Security 2014), pp. 719–732 (2014)
36. Zhang, D., Wang, Y., Suh, G.E., Myers, A.C.: A hardware design language for timing-sensitive information-flow security. ACM SIGPLAN Notices **50**(4), 503–516 (2015)

37. Zhang, T., Lee, R.B.: New models of cache architectures characterizing information leakage from cache side channels. In: Proceedings of the 30th Annual Computer Security Applications Conference, pp. 96–105 (2014)
38. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 305–316 (2012)