Cache Timing Side-Channel Vulnerability Checking with Computation Tree Logic

Shuwen Deng, Wenjie Xiong and Jakub Szefer
Yale University
New Haven, Connecticut
{shuwen.deng,wenjie.xiong,jakub.szefer}@yale.edu

ABSTRACT

Caches are one of the key features of modern processors as they help to improve memory access timing through caching recently used data. However, due to the timing differences between cache hits and misses, numerous timing side-channels have been discovered and exploited in the past. In this paper, Computation Tree Logic is used to model execution paths of the processor cache logic, and to derive formulas for paths that can lead to timing side-channel vulnerabilities. In total, 28 types of cache attacks are presented: 20 types of which map to attacks previously categorized or discussed in literature, and 8 types are new. Furthermore, to enable practical vulnerability checking, we present a new approach that limits the depth of the execution paths that need to be checked by the Computation Tree Logic, allowing for use of bounded model checking for Computation Tree Logic based cache security verification using the new three-step single-cache-block-access model.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; • **Theory of computation → Modal and temporal logics**; Verification by model checking;

KEYWORDS

timing side-channel, caches, Computation Tree Logic

ACM Reference Format:
Shuwen Deng, Wenjie Xiong and Jakub Szefer. 2018. Cache Timing Side-Channel Vulnerability Checking with Computation Tree Logic. In *HASP '18: Hardware and Architectural Support for Security and Privacy, June 2, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3214292.3214294

1 INTRODUCTION

With conventional processor caches, when a memory access is made, first, the memory address is checked to see if the data is in the cache, and if data is in the cache, it is called a cache *hit*, and data is returned from the cache quickly. If data is not in the cache, it is a cache *miss*, and main memory is accessed to retrieve the data, which takes longer amount of time. Because of this timing difference, it is possible to use timing of a program execution to determine if its memory accesses are hits or misses. Furthermore, when flushing or evicting a cache block, the timing of these operations depends on whether the cache block originally had valid data, which can also reveal the state of the cache block. The different timing information can then be used to leak secrets of a victim program, and this has been exploited by the various cache timing side-channel attacks, e.g., [1, 5, 7, 21, 30].

To address the security issues of cache timing side-channel attacks, numerous secure processor caches have been designed and analyzed through simulation and probability analysis in attempt to prove that the proposed caches indeed prevent attacks [12, 13, 26, 27, 33–36, 39, 40]. Furthermore, to reason about the different attacks, researchers have classified cache timing side-channels on basis whether the attacks leverage cache hits or misses and whether they are access-based or reuse-based, and whether the attacks leverage internal or external interference [22, 41]. Especially, [41] made use of finite-state machine to model cache architectures and used mutual information to calculate potential side-channel leakage of the modeled cache architectures. Meanwhile, [22] used probabilistic information flow graph to model interaction between a victim and an attacker program, and used probability of an attack success to evaluate how well different caches can defend against timing side-channel attacks.

In contrast to the existing work, this paper is the first work to explore Computation Tree Logic (CTL) [9, 10] to model execution paths of the processor cache logic, and to derive formulas for paths which indicate that there is a vulnerability to a potential attack. Rather than trying to model or simulate attackers, this work explicitly enumerates all the possible execution paths that could lead to an attack. CTL formulas are derived for these execution paths, and these formulas can be used with existing tools to check if a specific cache architecture is susceptible to one of the attacks. Especially, behavior of each single cache block can be modeled as a Kripke structure [23], which is used to describe the finite-state machine of the cache logic. Given the CTL formulas, they can be used to verify if there is at least one path in the computation tree derived from the Kripke structure that matches the formula, and if so, there is a potential vulnerability in the design.

The execution paths in computation tree of a cache could be potentially infinite. However, we show that any path corresponding to an attack can be represented as a path with three single-cache-block accesses steps (*step* 0, *step* 1, and *step* 2 in Section 4). While the computation tree is still large, it is now bounded in size and all the paths can be evaluated within this bounded computation tree to find potential vulnerabilities.

Through explicit enumeration of all the possible paths and their analysis, we have derived CTL formulas for 28 types of cache attacks. Of these attacks, 20 types are in agreement with the existing cache attack categorizations presented in [22, 27, 41] or correspond to known vulnerabilities. However, 8 of the types of attacks are previously not detailed in literature. The 28 CTL formulas can be applied to evaluate any cache architecture and can check, in bounded time, if the cache is vulnerable to an attack.

1.1 Contributions

This paper aims to help advance the field of verification of processor caches, and provides logic formulas that can be used to check

HASP '18, June 2, 2018, Los Angeles, CA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6500-0/18/06…$15.00
https://doi.org/10.1145/3214292.3214294

cache implementation against potential cache timing side-channel vulnerabilities. Our contributions are:

- First use of CTL to model execution paths of the processor cache logic focusing on side-channel attacks.
- Development of a new, three-step single-cache-block-access model for modeling all possible vulnerabilities that can lead to timing side-channel attacks in caches.
- Derivation of 28 types of cache side-channel attacks based on the three-step single-cache-block-access model, including 8 types of attacks previously not described in the existing literature in detail.
- Discussion of how to find cache timing side-channel vulnerabilities in real cache implementations by using the CTL formulas and bounded model checking.

2 COMPUTATION TREE LOGIC (CTL)

In this paper, we use Computation Tree Logic (CTL) [9, 10] to model execution paths of the processor cache logic, and to derive formulas for paths which can lead to attacks. CTL treats time as discrete and branching, where at each time step the system is in a defined state. It allows one to explore different execution paths that a system may go through as it executes. The number of the paths depends on the finite state machine describing the system. The lengths of the paths can be infinite, but bounded paths can be checked (c.f. bounded model checking [6]).

CTL is one type of temporal logic: temporal logic is the logic that contains any system of rules and symbolisms to represent, and reason about, propositions qualified in terms of time [14]. Propositions such as "there exists a path that property p holds starting at some time step until a step where property q holds" enable describing a system along with changes in its state over time.

There are also other forms of temporal logics, which include Linear Temporal Logic (LTL) [31] where at every moment in time there will only be one possible future that "actually takes place" [24]. Interval Temporal Logic (ITL) [2, 3] represents both propositional and first-order logical reasoning about periods of time. It is able to handle both sequential and parallel composition. Computation Tree Logic* (CTL*) [15] is a superset of Computation Tree Logic (CTL) and Linear Temporal Logic (LTL). There are also temporal logics for Hyperproperties [11], which are able to describe a set of trace properties and moreover can describe security policies such as noninterference. For this work, we use CTL as it allows for enumerating various execution paths and checking if there exists at least one path that matches a given formula.

2.1 Describing a System for CTL Checking

A system to be checked using CTL needs to be modeled as a finite-state machine, which can be described by a Kripke structure $M = <\mathbb{N}, I, \sigma, R>$ [23]. $\mathbb{N}$ is the finite set of states in the system and $I \subseteq \mathbb{N}$ is the set of initial states. $\mathscr{P}$ is the set of Boolean variables (labels), or some primitive propositions, and $\sigma : \mathbb{N} \rightarrow 2^{\mathscr{P}}$ is the mapping function of each state to the set of variables that are true in this state. Finally, $R \subseteq \mathbb{N} \times \mathbb{N}$ is the transition relation between states of the system.

Given the Kripke structure M and an initial state $s \in I$, the computation tree for the execution of the system can be derived starting in that state. The tree represents all the possible execution paths of the system based on all the possible states of the system, its inputs, and the state transitions.

A path from the root node, s, to a leaf node is called a path π. There are many paths in a tree. If a tree is unbounded, there could be infinite number of paths — later we explain how our work allows for use of bounded trees. If the tree is bounded, a path π_F will have finite number of states, m, and the finite sequence of states $s_0,\ s_1,\ \ldots,\ s_{m-1}$ of π_F hold that $\forall 0 \leqslant i < m,\ s_i \rightharpoonup s_{i+1}$ and $(s_i, s_{i+1}) \in R$. Paths are written as $\pi_F = s_0 \rightharpoonup s_1 \rightharpoonup s_2 \rightharpoonup \ \ldots \ \rightharpoonup s_{m-1}$, where s_i is $\pi_F[i]$, indicating the $i+1^{th}$ state in the path. Here $\sigma(\pi_F[i]) \in 2^{\mathscr{P}} (0 \leqslant i < m)$ represents the set of variables that are true in state $\pi_F[i]$, and $\rightharpoonup$ represents the predecessor to successor relation between states in a path.

Table 1: Symbols of CTL formula syntax.

Symbol	Description
$True$	logical true value
a	an atomic proposition
φ, φ_1, φ_2	valid CTL formulas
$\neg$	negation (unary op.)
$\wedge$	logical AND (binary op.)
$\vee$	logical OR (binary op.)
$\Rightarrow$	logical implication (binary op.)
$\Leftrightarrow$	logical bi-implication (if and only if, binary op.)
A	"for all" operator (along all paths, unary op.)
E	"exists" operator (along at least one path, unary op.)
X	"next" operator (unary op.)
G	"globally" operator (unary op.)
F	"eventually" operator (unary op.)
U	"until" operator (binary op.)

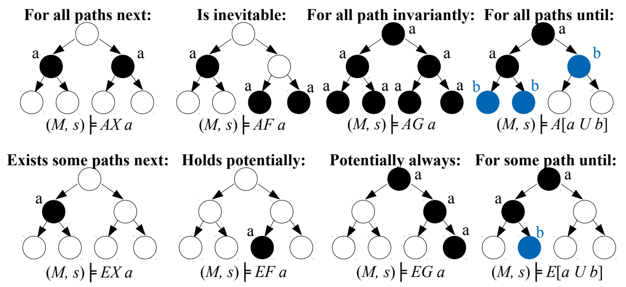

Figure 1: Example paths in computation trees that are true for the corresponding CTL formulas.

2.2 CTL Formula Syntax

CTL formulas can be described using the below syntax:

$$\begin{aligned} \varphi ::= & \ True|a|\neg\varphi|\varphi_1 \wedge \varphi_2|\varphi_1 \vee \varphi_2|\varphi_1 \Rightarrow \varphi_2|\varphi_1 \Leftrightarrow \varphi_2| \\ & AX\varphi|EX\varphi|AF\varphi|EF\varphi|AG\varphi|EG\varphi| \\ & A(\varphi_1 U \varphi_2)|E(\varphi_1 U \varphi_2) \end{aligned} \quad (1)$$

Symbols are explained in Table 1. The quantifiers over paths (A and E) are always combined with path-specific quantifiers (X, G, F and U). When evaluating CTL formulas, the unary operators bind stronger than the binary ones. Implication ($\Rightarrow$) and bi-implication ($\Leftrightarrow$) have the least precedence.

2.3 Semantics Over Paths

CTL formulas are true when there are certain execution paths in the computation tree for a system that matches that formula. Different CTL semantics, their names, and examples of paths that make such formulas true are shown in Figure 1. For example, if the CTL formula $(M,\ s) \models EF\varphi$ holds for a bounded computation tree of Kripke structure M, there exists at least one state in one of the paths in the tree where φ holds, i.e. φ "holds potentially" in the computation tree.

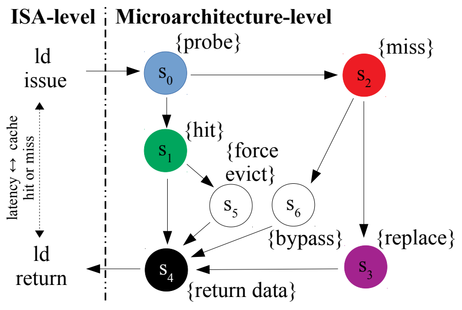

Figure 2: ISA and Microarchitecture level view of cache operation.

3 CACHES AND SIDE-CHANNEL ATTACKS

Operation of a processor cache is realized in the hardware cache controller logic, which can be described by a finite state machine. Each type of a processor cache has a finite number of states in its state machine and is deterministic[1]. The basic unit of a cache is a cache block. Based on the inputs (memory access requests) the state of each cache block evolves over time. Changes in the state of a cache block can be expressed as a path in CTL terminology. In this work, we show that CTL formulas can be used to describe execution paths which may lead to an attack; if for a particular cache state machine the CTL formula is never true, then the cache is secure against that type of attack.

3.1 Threat Model

To reason about cache attacks, we assume there is an attacker and a victim sharing the same cache. Third party interference has equivalent functionality to some victim's or attacker's behavior and is not considered. The attacker cannot directly access the state machine of the cache logic, but can make use of the timing difference between hits and misses to derive information about cache accesses by the victim – a timing side-channel. The attacker is able to observe its own timing and the timing of the operations of the victim. The attacker knows at least some of the source code of the victim (e.g. it can interpret some information from knowing a specific cache block access by a victim). The attacker is also able to force victim to execute a specific function (e.g. attacker can request victim to decrypt a specific piece of data).

3.2 Cache State Machine

A cache hit or miss has a one-to-one mapping to the state in the processor cache controller logic, and a hit or a miss has direct relation to the timing. Figure 2 shows the operation of a cache at two abstraction levels.

At the ISA level, memory access instructions, such as *ld* (load) and *st* (store), are used to access data. When the instruction is issued, data is requested from the memory subsystem. The cache block corresponding to the memory address is accessed. When the instruction is returned, the data is provided to the processor – for a memory access operation, the latency of the instruction has direct relation to whether it was a cache hit or cache miss. Similarly, for a flush or evict instruction, the latency of the operation reflects whether this cache block was an empty cache block (quick because there is nothing to flush or evict) or was not empty (slow because there was data to be flushed or evicted).

At the microarchitecture level, the cache controller is realized as a Kripke structure describing the finite state machine of the cache logic. A simplified structure is shown in Figure 2; a single memory access at the ISA level, corresponds to a set of state transitions at the microarchitecture level. The first state is s_0. Proposition "probe" holds in this state and is used to find out if the required data is in the cache or not. If it is, then s_1 state ("hit") is entered, followed by s_4 state ("return data"). If the data is not in the cache, then s_2 state ("miss") is entered, followed by s_3 state ("replace") where some other data is evicted and the requested data is brought into the cache. Finally, s_4 state ("return data") is entered and state will go back to s_0 ("probe") to wait for the next request.

Certain secure caches have other states, such as s_5 ("force evict") or s_6 ("bypass"), as shown in the state transition diagram in Figure 2. Other different possibie states exist as well. The goal of these extra states is to disrupt the timing of a cache *hit* and *miss*, thus eliminate the one-to-one correspondence between the timing and whether *hit* or *miss* state was entered.

[1]Certain secure caches use randomization to map addresses to cache blocks, but overall the operation of the cache is still deterministic.

3.3 Timing Side-Channel Attacks

Because of the time differences between hits and misses, conventional processor caches are susceptible to timing side-channel attacks, detailed in surveys such as [16, 28, 32, 42]. In a timing side-channel attack, there are an attacker, A and a victim, V. The goal for the attacker is to observe the timing of single cache block accesses of the victim or itself, and combined with some other knowledge, to determine what sensitive data the victim is operating on, e.g., guess bits of a secret key.

The attacker can themselves access a cache block by making single cache block accesses, or trigger the victim to make single cache block accesses, e.g. request victim to do some known computation, or both. The attacker usually knows what code the victim is executing, e.g. type of encryption algorithm, but does not know the victim's specific secrets. For instance, in the Prime + Probe attack [29], by priming to know some initial state of the cache, then letting victim execute, and finally observing the time of its own accessing to the same cache block, the attacker may extract some secret information.

4 VULNERABILITY MODELING

This paper proposes a new approach to reasoning about cache timing side-channel attacks by modeling execution paths that represent vulnerabilities to attacks as CTL formula in the form of:

$$(M,\ s) \models EF(E(E(step\ 0\ U\ step\ 1)\ U\ step\ 2))$$

The vulnerability checking can be done by applying such CTL formulas to a computation tree derived from a state machine of the cache controller, which will be discussed in Section 5.

In this section, we first introduce the three-step single-cache-block-access model and discuss the possibilities for each step in Section 4.1. Once all possibilities for each of the steps are described, then in Section 4.2, specific vulnerabilities and their CTL formulas for the three steps are derived. In Section 4.3, we show why the three-step model can model all possible n-step attacks.

4.1 Three-Step Single-Cache-Block-Access Model

For a vulnerability to exist, there needs to be an interference between the attacker's access and the victim's access to the same cache block, or an interference between two accesses of the victim to the same cache block. To be able to capture this, one needs to analyze different possible accesses to a single cache block, denoted as "single-cache-block accesses". Furthermore, at least three such accesses, or "steps", are needed to model the cache timing side-channel attacks.

Table 2: Six possible conditions for state of a single cache block in the three-step single-cache-block-access modeling procedure.

Condition	Description
V_1/A_1	A specific memory location of the victim or attacker is brought into the single cache block targeted by and known to attacker.
V_x	A piece of memory containing data from a range of victim's memory addresses is accessed. Attacker knows the range, but not specific addresses accessed. Therefore, the targeted cache block is potentially accessed by V_x.
V_R/A_R	Victim or attacker does single-cache-block access to "remove" the cache block contents so neither attacker's known data nor victim's data is in the cache. It can be achieved by using other cache block accessing to evict the original data, or using flush instructions like $cflush$, or can be achieved using cache coherence protocol.
$\star$	Attacker has no knowledge about memory location in this cache block.

First, some memory operation is performed that sets one single cache block in some known initial state, which is $step$ 0. Then some action can be done by the victim or attacker, which is $step$ 1, and final action is taken to derive information by observing timing, which is $step$ 2. The attacker should be able to interpret the victim's behavior in $step$ 2, if there is a possible attack.

We identify six possible conditions of a cache block in Table 2. V_1 requires attacker to drive the victim to access specific memory location known by the attacker and put it into the cache block, while A_1 means the attacker directly accessing a specific memory address themselves and putting data at that location. In both cases attacker knows specific address of memory location in the cache block. V_x is more general than V_1, where victim accesses one of multiple security critical memory addresses (e.g. one of AES S-Box table entries) and put the data into the cache block, but which specific memory address or which cache block was accessed is not known. Therefore, potentially the cache block targeted on could be accessed by V_x. V_R or A_R requires attacker or victim to remove (clear or evict) data from a cache block so neither attacker's known data nor victim's data is in the cache. It can be achieved by using other cache block accesses to evict the original cache block, or by using flush instructions like $cflush$, or by using cache coherence protocol among different cache levels or even different CPU cores. $\star$ means the initial contents of the cache block is unknown to attacker.

4.1.1 Modeling Step 0. The initial state of a cache block can be modeled as $step$ 0. Any of the six conditions in Table 2 can be achieved by performing one valid single-cache-block access operation by the attacker or victim.

4.1.2 Modeling Step 1. Once the cache block is in a known state, for an attack to exist, some interference needs to be created. This is modeled by $step$ 1. There are five possible conditions in this step: A_R, V_R, A_1, V_1 or V_x. $\star$ is excluded as it cannot lead to an attack – putting the cache block in an unknown state removes useful information.

4.1.3 Modeling Step 2. The final step in any attack is to make a cache block access to try to observe whether any interference happened. Here possible conditions are A_R, V_R, A_1, V_1 or V_x as the attacker needs to effectively access a memory location to observe its own fast / slow timing, resulting from whether it is an empty cache block when trying to remove data (for data removal) or whether it is a cache hit / miss (for data access), or the victim is driven to do some cache block access to put corresponding data or remove data from cache block so attacker can observe victim's fast / slow timing. If the observed timing of $Step$ 2 for one single cache block is always fast or always slow, attacker cannot extract information from (the lack of) timing change. Therefore, timing change is a requirement for effective attack, but it is not sufficient. $\star$ would not give useful information and is not considered for this step.

4.1.4 Deriving CTL Formula From the Three Steps. Based on the three steps, CTL formula are derived in the form of: $(M,\ s) \models EF(E(E(step\ 0\ U\ step\ 1)\ U\ step\ 2))$. The possible states for the steps are inserted in the place of "step 0", "step 1", and "step 2". For example, if the states in the conditions are A_1, V_x, and A_1, then the resulting formula is: $(M,\ s) \models EF(E(E(A_1\ U\ V_x)\ U\ A_1))$.

4.2 Formulas for Attacks

Table 3 shows exhaustive list of all possible three-step combinations of conditions for one single cache block. As discussed in the prior Section, $step$ 0 can be $\star$, A_R, V_R , A_1, V_1 and V_x, $step$ 1 and $step$ 2 can be A_R, V_R, A_1, V_1 and V_x. Based on these, all possible three-step CTL formulas can be obtained. In Table 3, the three $step$ columns give the conditions for each evaluated path. Observed Timing of $step$ 2 column represents possible fast or slow timing information. When loading data in $step$ 2, fast load time corresponds to cache hit and slow load time corresponds to cache miss. For a cache block data removal (A_R or V_R), fast data removal time indicates that the corresponding cache block previously did not have data and slow data removal time indicates that the corresponding cache block previously had data. Possible Attack column indicates if the three steps in the corresponding row represent vulnerability to an attack. Categorization column gives a name to the attack if such exists in a row. Finally, footnote explanations give some intuition why, or why not, the three steps in a table row correspond, or do not correspond, to an attack.

Through evaluation of these exhaustive combinations, we have found in total 28 kinds of vulnerabilities. Of the 28 types of attacks presented in this paper, 20 of them have example attacks, as shown in Table 4. However, the existing example attacks do not cover all possible attacks in some types. For instance, Type M vulnerability can be implemented as Flush + Flush attack [18]. But other remain-to-be-discovered attacks are also possible, e.g., Flush + Evict attack, Evict + Evict attack, etc. Moreover, 8 of the attacks are new. Combined by some common features, these 8 new vulnerabilities are explained below.

4.2.1 Type A, B Attacks. For Type A, B attacks, the victim fails to reuse its own security critical memory location in Step 2 because there is intermediate access that removes the data from the cache block. This implies the contention between different, known memory locations from which data is removed and unknown victim's memory location for the same cache block.

- Victim: performs memory access to put some security critical memory location into the cache block.
- Attacker: removes the cache block's data (Type A) or lets the victim remove the cache block's data (Type B).
- Victim: performs memory access to put some security critical memory location into the cache block again and the attacker can observe the victim's data load time. (Longer time indicates this removed cache block's memory location of the attacker has the same index as the unknown memory location of the victim.)

4.2.2 Type C, D, E, F Attacks. In the Type C, D, E, F attacks, victim experiences cache hit due to attacker's previous cache block access in $step$ 1, which leads to decreased cache access time. This time difference is observed by the attacker.

Table 3: Exhaustive list of all potential three-step single-cache-block accesses for cache timing side-channel vulnerability analysis.

Num.	*Step* 0	*Step* 1	*Step* 2	Observed Timing of *Step* 2	Possible Attack	Catego-rization	Num.	*Step* 0	*Step* 1	*Step* 2	Observed Timing of *Step* 2	Possible Attack	Catego-rization	Num.	*Step* 0	*Step* 1	*Step* 2	Observed Timing of *Step* 2	Possible Attack	Catego-rization
1	$\star$	A_R	A_R	fast	N[1]	—	51	V_R	V_R	V_1	slow	N[2]	—	101	V_1	V_1	V_R	slow	N[3]	—
2	A_R	A_R	A_R	fast	N[1]	—	52	A_1	V_R	V_1	slow	N[2]	—	102	V_x	V_1	V_R	slow	N[3]	—
3	V_R	A_R	A_R	fast	N[1]	—	53	V_1	V_R	V_1	slow	N[2]	—	103	$\star$	V_1	A_1	fast	N[4]	—
4	A_1	A_R	A_R	fast	N[1]	—	54	V_x	V_R	V_1	slow	N[2]	—	104	A_R	V_1	A_1	fast	N[4]	—
5	V_1	A_R	A_R	fast	N[1]	—	55	$\star$	V_R	V_x	fast/slow	N[5]	—	105	V_R	V_1	A_1	fast	N[4]	—
6	V_x	A_R	A_R	fast	N[1]	—	56	A_R	V_R	V_x	fast/slow	N[5]	—	106	A_1	V_1	A_1	fast	N[4]	—
7	$\star$	A_R	V_R	fast	N[1]	—	57	V_R	V_R	V_x	fast/slow	N[5]	—	107	V_1	V_1	A_1	fast	N[4]	—
8	A_R	A_R	V_R	fast	N[1]	—	58	A_1	V_R	V_x	fast/slow	N[5]	—	108	V_x	V_1	A_1	fast	N[4]	—
9	V_R	A_R	V_R	fast	N[1]	—	59	V_1	V_R	V_x	fast/slow	N[5]	—	109	$\star$	V_1	V_1	fast	N[4]	—
10	A_1	A_R	V_R	fast	N[1]	—	60	V_x	V_R	V_x	fast/slow	Y[7]	Type B	110	A_R	V_1	V_1	fast	N[4]	—
11	V_1	A_R	V_R	fast	N[1]	—	61	$\star$	A_1	A_R	slow	N[3]	—	111	V_R	V_1	V_1	fast	N[4]	—
12	V_x	A_R	V_R	fast	N[1]	—	62	A_R	A_1	A_R	slow	N[3]	—	112	A_1	V_1	V_1	fast	N[4]	—
13	$\star$	A_R	A_1	slow	N[2]	—	63	V_R	A_1	A_R	slow	N[3]	—	113	V_1	V_1	V_1	fast	N[4]	—
14	A_R	A_R	A_1	slow	N[2]	—	64	A_1	A_1	A_R	slow	N[3]	—	114	V_x	V_1	V_1	fast	N[4]	—
15	V_R	A_R	A_1	slow	N[2]	—	65	V_1	A_1	A_R	slow	N[3]	—	115	$\star$	V_1	V_x	fast/slow	N[5]	—
16	A_1	A_R	A_1	slow	N[2]	—	66	V_x	A_1	A_R	slow	N[3]	—	116	A_R	V_1	V_x	fast/slow	Y[6]	Type H
17	V_1	A_R	A_1	slow	N[2]	—	67	$\star$	A_1	V_R	slow	N[3]	—	117	V_R	V_1	V_x	fast/slow	Y[6]	Type I
18	V_x	A_R	A_1	slow	N[2]	—	68	A_R	A_1	V_R	slow	N[3]	—	118	A_1	V_1	V_x	fast/slow	Y[6]	Type J
19	$\star$	A_R	V_1	slow	N[2]	—	69	V_R	A_1	V_R	slow	N[3]	—	119	V_1	V_1	V_x	fast/slow	Y[6]	Type K
20	A_R	A_R	V_1	slow	N[2]	—	70	A_1	A_1	V_R	slow	N[3]	—	120	V_x	V_1	V_x	fast/slow	Y[7]	Type L
21	V_R	A_R	V_1	slow	N[2]	—	71	V_1	A_1	V_R	slow	N[3]	—	121	$\star$	V_x	A_R	fast/slow	N[5]	—
22	A_1	A_R	V_1	slow	N[2]	—	72	V_x	A_1	V_R	slow	N[3]	—	122	A_R	V_x	A_R	fast/slow	Y[7]	Type M
23	V_1	A_R	V_1	slow	N[2]	—	73	$\star$	A_1	A_1	fast	N[4]	—	123	V_R	V_x	A_R	fast/slow	Y[7]	Type N
24	V_x	A_R	V_1	slow	N[2]	—	74	A_R	A_1	A_1	fast	N[4]	—	124	A_1	V_x	A_R	slow	N[8]	—
25	$\star$	A_R	V_x	fast/slow	N[5]	—	75	V_R	A_1	A_1	fast	N[4]	—	125	V_1	V_x	A_R	slow	N[8]	—
26	A_R	A_R	V_x	fast/slow	N[5]	—	76	A_1	A_1	A_1	fast	N[4]	—	126	V_x	V_x	A_R	fast/slow	Y[7]	Type O
27	V_R	A_R	V_x	fast/slow	N[5]	—	77	V_1	A_1	A_1	fast	N[4]	—	127	$\star$	V_x	V_R	fast/slow	N[5]	—
28	A_1	A_R	V_x	fast/slow	N[5]	—	78	V_x	A_1	A_1	fast	N[4]	—	128	A_R	V_x	V_R	fast/slow	Y[7]	Type P
29	V_1	A_R	V_x	fast/slow	N[5]	—	79	$\star$	A_1	V_1	fast	N[4]	—	129	V_R	V_x	V_R	fast/slow	Y[7]	Type Q
30	V_x	A_R	V_x	fast/slow	Y[7]	Type A	80	A_R	A_1	V_1	fast	N[4]	—	130	A_1	V_x	V_R	slow	N[8]	—
31	$\star$	V_R	A_R	fast	N[1]	—	81	V_R	A_1	V_1	fast	N[4]	—	131	V_1	V_x	V_R	slow	N[8]	—
32	A_R	V_R	A_R	fast	N[1]	—	82	A_1	A_1	V_1	fast	N[4]	—	132	V_x	V_x	V_R	fast/slow	Y[7]	Type R
33	V_R	V_R	A_R	fast	N[1]	—	83	V_1	A_1	V_1	fast	N[4]	—	133	$\star$	V_x	A_1	fast/slow	N[5]	—
34	A_1	V_R	A_R	fast	N[1]	—	84	V_x	A_1	V_1	fast	N[4]	—	134	A_R	V_x	A_1	fast/slow	Y[6]	Type S
35	V_1	V_R	A_R	fast	N[1]	—	85	$\star$	A_1	V_x	fast/slow	N[5]	—	135	V_R	V_x	A_1	fast/slow	Y[6]	Type T
36	V_x	V_R	A_R	fast	N[1]	—	86	A_R	A_1	V_x	fast/slow	Y[6]	Type C	136	A_1	V_x	A_1	fast/slow	Y[7]	Type U
37	$\star$	V_R	V_R	fast	N[1]	—	87	V_R	A_1	V_x	fast/slow	Y[6]	Type D	137	V_1	V_x	A_1	fast/slow	Y[7]	Type V
38	A_R	V_R	V_R	fast	N[1]	—	88	A_1	A_1	V_x	fast/slow	Y[6]	Type E	138	V_x	V_x	A_1	fast/slow	Y[6]	Type W
39	V_R	V_R	V_R	fast	N[1]	—	89	V_1	A_1	V_x	fast/slow	Y[6]	Type F	139	$\star$	V_x	V_1	fast/slow	N[5]	—
40	A_1	V_R	V_R	fast	N[1]	—	90	V_x	A_1	V_x	fast/slow	Y[7]	Type G	140	A_R	V_x	V_1	fast/slow	Y[6]	Type X
41	V_1	V_R	V_R	fast	N[1]	—	91	$\star$	V_1	A_R	slow	N[3]	—	141	V_R	V_x	V_1	fast/slow	Y[6]	Type Y
42	V_x	V_R	V_R	fast	N[1]	—	92	A_R	V_1	A_R	slow	N[3]	—	142	A_1	V_x	V_1	fast/slow	Y[7]	Type Z
43	$\star$	V_R	A_1	slow	N[2]	—	93	V_R	V_1	A_R	slow	N[3]	—	143	V_1	V_x	V_1	fast/slow	Y[7]	Type AA
44	A_R	V_R	A_1	slow	N[2]	—	94	A_1	V_1	A_R	slow	N[3]	—	144	V_x	V_x	V_1	fast/slow	Y[6]	Type AB
45	V_R	V_R	A_1	slow	N[2]	—	95	V_1	V_1	A_R	slow	N[3]	—	145	$\star$	V_x	V_x	fast	N[4]	—
46	A_1	V_R	A_1	slow	N[2]	—	96	V_x	V_1	A_R	slow	N[3]	—	146	A_R	V_x	V_x	fast	N[4]	—
47	V_1	V_R	A_1	slow	N[2]	—	97	$\star$	V_1	V_R	slow	N[3]	—	147	V_R	V_x	V_x	fast	N[4]	—
48	V_x	V_R	A_1	slow	N[2]	—	98	A_R	V_1	V_R	slow	N[3]	—	148	A_1	V_x	V_x	fast	N[4]	—
49	$\star$	V_R	V_1	slow	N[2]	—	99	V_R	V_1	V_R	slow	N[3]	—	149	V_1	V_x	V_x	fast	N[4]	—
50	A_R	V_R	V_1	slow	N[2]	—	100	A_1	V_1	V_R	slow	N[3]	—	150	V_x	V_x	V_x	fast	N[4]	—

[1] Observed timing of *step* 2's data removal will always be fast, because data in this cache block is already removed in *Step* 1. [2] Observed timing of *step* 2's data accessing will always be slow, because data in this cache block is removed in *Step* 1.
[3] Observed timing of *step* 2's data removal will always be slow, because data in this cache block is accessed in *Step* 1. [4] Observed timing of *step* 2's data accessing will always be fast, because data in this cache block is already accessed in *Step* 1.
[5] Victim's behavior from timing observation cannot be interpreted because there are different possibilities existing for different subsets of *Step* 0. [6] Observed timing of *Step* 2 depends on whether victim's unknown address in *Step* 1 or *Step* 2 maps to the same cache block as attacker's known address. Fast timing means the same cache block is mapped, while slow timing means the opposite situation. [7] Observed timing of *Step* 2 depends on whether victim's unknown address in *Step* 1 or *Step* 2 have the same index in the cache as attacker's or victim's known address. Slow timing means they have the same index, while fast timing means the opposite situation. [8] Observed timing of *step* 2's data removal will always be slow, because data in this cache block is accessed in *Step* 0 no matter what *step* 1's V_x address is.

Table 4: Different types of cache timing side-channel attack vulnerabilities extracted from all possible combinations listed in Table 3 and their relations with existing attack categorizations or known attack examples.

CTL Format of the Attack	Categorization in this paper	Categorization in [22][27]	Categorization in [41]	Known Attack Example
$EF(E(E(V_x\ U\ A_R)\ U\ V_x))$	Type A	–	–	
$EF(E(E(V_x\ U\ V_R)\ U\ V_x))$	Type B	–	–	
$EF(E(E(A_R\ U\ A_1)\ U\ V_x))$	Type C	–	–	
$EF(E(E(V_R\ U\ A_1)\ U\ V_x))$	Type D	–	–	
$EF(E(E(A_1\ U\ A_1)\ U\ V_x))$	Type E	–	–	
$EF(E(E(V_1\ U\ A_1)\ U\ V_x))$	Type F	–	–	
$EF(E(E(V_x\ U\ A_1)\ U\ V_x))$	Type G	Type 1	–	(1)
$EF(E(E(A_R\ U\ V_1)\ U\ V_x))$	Type H	Type 3	Type IV	(2)
$EF(E(E(V_R\ U\ V_1)\ U\ V_x))$	Type I	Type 3	Type IV	(2)
$EF(E(E(A_1\ U\ V_1)\ U\ V_x))$	Type J	Type 3	Type IV	(2)
$EF(E(E(V_1\ U\ V_1)\ U\ V_x))$	Type K	Type 3	Type IV	(2)
$EF(E(E(V_x\ U\ V_1)\ U\ V_x))$	Type L	–	Type II	(3)
$EF(E(E(A_R\ U\ V_x)\ U\ A_R))$	Type M	–	–	(4)
$EF(E(E(V_R\ U\ V_x)\ U\ A_R))$	Type N	–	–	(4)
$EF(E(E(V_x\ U\ V_x)\ U\ A_R))$	Type O	–	–	(4)
$EF(E(E(A_R\ U\ V_x)\ U\ V_R))$	Type P	–	–	(4)
$EF(E(E(V_R\ U\ V_x)\ U\ V_R))$	Type Q	–	–	(4)
$EF(E(E(V_x\ U\ V_x)\ U\ V_R))$	Type R	–	–	(4)
$EF(E(E(A_R\ U\ V_x)\ U\ A_1))$	Type S	Type 4	Type III	(5)
$EF(E(E(V_R\ U\ V_x)\ U\ A_1))$	Type T	Type 4	Type III	(5)
$EF(E(E(A_1\ U\ V_x)\ U\ A_1))$	Type U	Type 2	Type I	(6)
$EF(E(E(V_1\ U\ V_x)\ U\ A_1))$	Type V	–	–	
$EF(E(E(V_x\ U\ V_x)\ U\ A_1))$	Type W	Type 4	Type III	(5)
$EF(E(E(A_R\ U\ V_x)\ U\ V_1))$	Type X	Type 3	Type IV	(2)
$EF(E(E(V_R\ U\ V_x)\ U\ V_1))$	Type Y	Type 3	Type IV	(2)
$EF(E(E(A_1\ U\ V_x)\ U\ V_1))$	Type Z	–	–	
$EF(E(E(V_1\ U\ V_x)\ U\ V_1))$	Type AA	–	Type II	(3)
$EF(E(E(V_x\ U\ V_x)\ U\ V_1))$	Type AB	Type 3	Type IV	(2)

(1) Evict + Time attack [29]. (4) Flush + Flush attack [18].
(2) Cache Collosion attack [7]. (5) Flush + Reload attack [37, 38], Evict + Reload attack [19].
(3) Bernstein's attack [5]. (6) Prime + Probe attack [29, 30], Alias-driven attack [20].

- Attacker: removes the cache block's data (Type C) or lets the victim remove the cache block's data (Type D) or accesses a cache block at a known memory location (Type E) or drives the victim to access a cache block at known memory location (Type F).
- Attacker: accesses the cache block with known memory location again.
- Victim: performs memory access to put some security critical memory location into the cache block and the attacker can observe the victim's data load time. (Shorter data access time indicates victim's address of security critical memory location maps to the same cache block as attacker's known memory location.)

4.2.3 Type V Attack. In this attack, there is contention between known memory address of the victim and unknown victim's address for the same cache block:

- Victim: performs memory access to put corresponding memory location known to the attacker into the cache block.
- Victim: performs memory access to put some security critical memory location into the cache block.
- Attacker: accesses the same memory location as victim's known address, observes if there is a hit or a miss due to contention with victim's prior accesses. (A miss indicates attacker's memory location has the same index as unknown victim's security critical memory access.)

4.2.4 Type Z Attack. In the Type Z attack, the victim fails to reuse attacker's same known initial memory location because of the intermediate unknown victim's memory access. This implies the contention between different known memory location and unknown victim's memory location for the same cache block.

- Attacker: performs memory access to put corresponding memory location known to itself into the cache block.
- Victim: performs memory access to put some security critical memory location into the cache block.
- Victim: accesses the cache block at the same location as the attacker; attacker observes the timing to derive victim's hit or miss information. (Longer time indicates this known victim's memory location has the same index as the unknown memory location of the victim.)

4.3 Analysis of Three-Step Single-Cache-Block-Access Model

In the following discussion, we will analyze why three-step single-cache-block-access model can cover all possible cache timing side-channel vulnerabilities based on our threat model, while less-than-three-step model is inadequate, and why using more steps is not necessary. Let β denotes the number of single-cache-block accesses in one attack, α denotes the number of single-cache-block accesses in a model to model all possible attacks. $\beta,\ \alpha,\ n \in \mathbb{Z}^+$.

We first show the model needs at least three accesses to model all possible attacks ($\alpha > 2$) by proof of contradiction. If $\alpha \leqslant 2$, the model with α accesses cannot model Type A vulnerability, which is required to have three single-cache-block accesses by victim, attacker and victim, respectively. This contradicts the assumption that a model with α accesses can model all attacks. Therefore, $\alpha > 2$.

We want to show the model with $\alpha = 3$ accesses can model all possible attacks with any β accesses. (i) For $\beta = 1$, interference between attacker's and victim's access to a cache block, or between two accesses of the victim is not possible, thus no attack can be achieved by one access. (ii) For $\beta = 2$, we can use the three-step model by setting $Step$ 0 to be $\star$ – this first single-cache-block access gives no information and thus the next two steps form a two-step single-cache-block accesses. As shown in the exhaustive lists of Table 3, there is no attacks with $Step$ 0 being $\star$. (iii) For $\beta = \alpha = 3$, the model has the same number of steps as the attack. Table 3 gives exhaustive list of all possible three-step single-cache-block accesses and shows that there are 28 types of attacks, thus β can be 3 and $\alpha = 3$ can cover this condition. (iv) For $\beta > 3$, let β^* denote the number of accesses in a cache access sequence that has the property: If the sequence of β access can form an attack, the sequence of β^* access must also be an attack. The sequence of β-step single-cache-block accesses can be reduced to β^*-step accesses based on following rules:

(1) If single cache block accesses have a sub-pattern such as $\{\ldots \rightsquigarrow \star \rightsquigarrow \ldots\}$, they can be divided into two separate parts, based on the position of $\star$. The original β^*-step single-cache-block-access model can then be reduced by at least 1 step for each of the two separate patterns. Each pattern can be recursively analyzed again.
(2) If the remaining single cache block accesses have a pattern such as $\{\ldots \rightsquigarrow A_R \rightsquigarrow A_R \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow A_R \rightsquigarrow V_R \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow V_R \rightsquigarrow A_R \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow V_R \rightsquigarrow V_R \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow A_1 \rightsquigarrow A_1 \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow A_1 \rightsquigarrow V_1 \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow V_1 \rightsquigarrow A_1 \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow V_1 \rightsquigarrow V_1 \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow V_x \rightsquigarrow V_x \rightsquigarrow \ldots\}$, due to the repeat single cache block location accessed, they can be reduced to $\{\ldots \rightsquigarrow A_R \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow V_R \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow A_R \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow V_R \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow A_1 \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow V_1 \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow A_1 \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow V_1 \rightsquigarrow \ldots\}$, $\{\ldots \rightsquigarrow V_x \rightsquigarrow \ldots\}$, respectively. Therefore, β^* can be reduced to $(\beta^* - 1)$. Each pattern can be recursively analyzed again.
(3) Following the above two rules, either $\beta^* \leqslant 3$ holds, or the following sub-patterns in the single-cache-block access pattern

still exist: $\{\ldots \rightsquigarrow (A_R/V_R/A_1/V_1) \rightsquigarrow V_x \rightsquigarrow (A_R/V_R/A_1/V_1) \rightsquigarrow \ldots\}$ or $\{\ldots \rightsquigarrow V_x \rightsquigarrow (A_R/V_R/A_1/V_1) \rightsquigarrow V_x \rightsquigarrow \ldots\}$. These two patterns map to known vulnerabilities listed in Table 4: Type M, N, P, Q, S, T, U, V, X, Y, Z, AA for the first sub-pattern (Except 4 combinations derived from $\{\ldots \rightsquigarrow (A_1/V_1) \rightsquigarrow V_x \rightsquigarrow (A_R/V_R) \rightsquigarrow \ldots\}$ where slow data removal time is certain because of *Step* 0's known data accessing and timing observation of A_R/V_R is always slow), and Type A, B, G and L for the second sub-pattern. Therefore, the longer pattern of multiple cache block accesses will always be matched by these known three-step vulnerability patterns: β^* can be always reduced to less than or equal to three steps, which can be covered by α-step modeling where α equals to 3.

In conclusion, $\alpha = 3$, which shows three-step single-cache-block-access model can cover all possible timing cache side-channel vulnerabilities based on our threat model and is the most simplified model. At the same time, modeling paths having more than three steps can be reduced to three steps only.

5 VULNERABILITY CHECKING

Given the CTL formulas for potential vulnerabilities, they need to be applied to a cache for actual verification of the given cache design. The complexity of the checking depends on the cache architecture. Core ideas of the vulnerability checking in this Section are:

- Modeling state transitions of each cache block as a computation tree derived from the Kripke structure based on the cache controller logic.
- Bounding number of states of the execution paths in the computation tree according to the three-step model.
- Mapping each step of three-step single-cache-block-access model to the states in the computation tree and check whether the CTL formulas for each attack in Table 4 hold or not.

5.1 Bounded Checking with Three-Step Single-Cache-Block-Access Rule

Figure 3 shows a simplified cache state machine (described as a Kripke structure) targeting on only one single cache block and using "ld" instruction accessing as the example. It follows Figure 2, but with s_5 ("force evict") and s_6 ("bypass") states omitted. For each element of the Kripke structure $M = <\mathbb{N}, I, \sigma, R>$, $\mathbb{N} = \{s_0, s_1, s_2, s_3, s_4\}$, $I = s_0$, $\sigma = \{s_0 \rightarrow \{\text{probe}\}, s_1 \rightarrow \{\text{hit}\}, s_2 \rightarrow \{\text{miss}\}, s_3 \rightarrow \{\text{replace}\}, s_4 \rightarrow \{\text{return data}\}\}$, $R = \{s_0 \rightharpoonup s_1, s_0 \rightharpoonup s_2, s_1 \rightharpoonup s_4, s_2 \rightharpoonup s_3, s_3 \rightharpoonup s_4, s_4 \rightharpoonup s_0\}$. The cache starts in s_0 state ("probe") – consequently the computation tree is rooted at the s_0 state. From there, for all possible addresses of data in the cache block (for the Figure example addresses are 1 bit, so there are 2 possible values) and for all the possible memory requests (again, two possibilities for the address in Figure 3), a computation tree is built by enumerating all the possible transitions. The computation tree for the example is shown on the right-hand side of Figure 3.

From the s_0 state ("probe") the tree is built, and different states are explored in the execution paths. Eventually, each path will reach the s_4 state ("return data"). This is equivalent to having finished one single-cache-block-access operation. For our proposed modeling, total of three operations are needed, thus each path is explored further. From s_4 state ("return data") the s_0 state ("probe") is visited, and again all the possible paths inputs are considered. Note, this time the data in the cache block is fixed (it is the data in the block at the prior s_4 state ("return data")) so only different cache inputs are considered but not different states of that block. Going forward, each path will have s_4 state ("return data") again (end of second single-cache-block access). The tree is further expanded until third

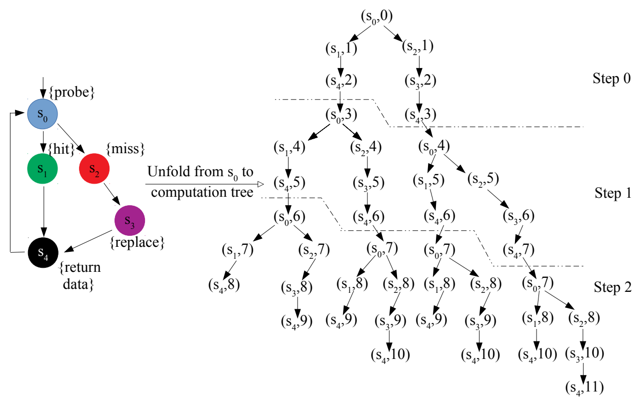
{probe}
s_0
{hit}
{miss}
s_1
s_2
s_3
{replace}
s_4
{return data}
Unfold from s_0 to computation tree
$(s_0,0)$
$(s_1,1)$ $(s_2,1)$
$(s_4,2)$ $(s_3,2)$
Step 0
$(s_0,3)$ $(s_4,3)$
$(s_1,4)$ $(s_2,4)$ $(s_0,4)$
$(s_4,5)$ $(s_3,5)$ $(s_1,5)$ $(s_2,5)$
Step 1
$(s_0,6)$ $(s_4,6)$ $(s_4,6)$ $(s_3,6)$
$(s_1,7)$ $(s_2,7)$ $(s_0,7)$ $(s_0,7)$ $(s_4,7)$
$(s_4,8)$ $(s_3,8)$ $(s_1,8)$ $(s_2,8)$ $(s_1,8)$ $(s_2,8)$ $(s_0,7)$
Step 2
$(s_4,9)$ $(s_4,9)$ $(s_3,9)$ $(s_4,9)$ $(s_3,9)$ $(s_1,8)$ $(s_2,8)$
$(s_4,10)$ $(s_4,10)$ $(s_4,10)$ $(s_3,10)$
$(s_4,11)$

Figure 3: Simplified sample cache controller Kripke structure, following Figure 2, but with the s_5 state (*force evict*) and s_6 state (*bypass*) omitted; and the resulting computation tree of three-step single-cache-block-access model. For convenience, each node in the tree is composed of a pair indicating the state in the Kripke structure and the level of the node in the tree.

s_4 state ("return data") is reached on each path. Different caches, especially secure caches, may have many different intermediate states so the tree can become quite complicated, but will have similar structure and each path will be bounded.

Based on the discussion of Section 4, known cache side-channel vulnerabilities can be represented by the three-step model. Thus, checking states of three single-cache-block accesses is enough to verify different possibilities of cache timing side-channel vulnerabilities for different caches.

5.2 Secure Cache Model Checking with CTL

In order to prevent different types of cache timing side-channel vulnerabilities listed in Table 4, the designs of secure caches should not allow paths which correspond to the vulnerability to exist in the state transition of cache's computation tree unfolded from cache's Kripke structure.

When a secure cache is implemented and a computation tree similarly as right-hand side of Figure 3 is derived from the corresponding Kripke structure, in order to check formula derived from Section 4 with each step under the condition from Table 2, primitive propositions ($2^{\mathscr{P}}$) for each state should have: address of the data in the cache block, whom the addresses belong to (victim or attacker) and condition of that block ("probe", "hit", "miss", "flushing", etc.). Other primitive propositions should be removed using predicate abstraction [17] to simplify the state machine. With the above information for each state, a mapping function from the primitive propositions to conditions described in Table 2 can be derived and which condition one state corresponds to can be obtained. Each of the 28 types of cache timing side-channel vulnerabilities in the form of CTL formula will then be checked one by one based on the mapping function to see if the represented vulnerabilities exist in the computation tree for each single cache block of the secure cache designs.

5.3 Towards Verification of Secure Caches

As the number of states in each path of the tree is finite, it is possible to use Bounded Modeling Checking (BMC) [6] for the vulnerability checking. There are existing tools that allow for performing bounded modeling checking for CTL logic. UPPAAL [25] is an integrated tool environment to model, simulate and verify real-time embedded systems. RuleBase [4] is an industry-oriented formal verification tool to formally verify critical portions of hardware designs. NuSMV 2 [8] takes SMV modeling language with CTL specifications to do model checking.

Our ongoing work is on implementation of various secure cache designs and using the tools listed above to check for existence of potential vulnerabilities in the different cache designs. We expect the three-step bounded single-cache-block-access model will allow for fast, practical verification of cache architectures.

6 CONCLUSION

Cache timing side-channel vulnerabilities based on cache access and timing differences are getting more and more attention and represent an increasing threat. In this work, we use Computation Tree Logic to model execution paths of the processor cache logic and derive Computation Tree Logic formulas representing vulnerabilities to cache attacks. Based on the study of cache timing side-channel vulnerabilities, we propose that for existing cache timing side-channel vulnerabilities, a three-step single-cache-block-access model is able to cover all the possibilities for potential attacks. We presented 28 types of vulnerabilities based enumeration of all potential three-step execution paths, 20 of which map to the known attack categories or have been previously demonstrated, while 8 of which are new. We also show how existing tools for bounded model checking can be used with our approach and can help with security verification of processor caches.

7 ACKNOWLEDGEMENT

We would like to thank Professor Ruzica Piskac and Mr. Mark Santolucito for helpful discussions, and the anonymous reviewers for their feedback on this paper. This work was supported in part by the National Science Foundation's grant number 1524680.

REFERENCES

[1] Onur Acıiçmez and Çetin Kaya Koç. 2006. Trace-driven cache attacks on AES (short paper). In *International Conference on Information and Communications Security*. Springer, 112–121.
[2] James F Allen. 1984. Towards a general theory of action and time. *Artificial intelligence* 23, 2 (1984), 123–154.
[3] James F Allen. 1990. Maintaining knowledge about temporal intervals. In *Readings in qualitative reasoning about physical systems*. Elsevier, 361–372.
[4] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. 1996. RuleBase: An industry-oriented formal verification tool. In *Proceedings of the 33rd annual Design Automation Conference*. ACM, 655–660.
[5] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).
[6] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 193–207.
[7] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 201–215.
[8] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*. Springer, 359–364.
[9] Edmund M Clarke and E Allen Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*. Springer, 52–71.
[10] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 2 (1986), 244–263.
[11] Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. 2014. Temporal logics for hyperproperties. In *International Conference on Principles of Security and Trust*. Springer, 265–284.
[12] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation.. In *USENIX Security Symposium*. 857–874.
[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 (TACO)* 8, 4 (2012), 35.
[14] E Allen Emerson. 1990. Temporal and modal logic. In *Formal Models and Semantics*. Elsevier, 995–1072.
[15] E Allen Emerson and Joseph Y Halpern. 1986. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)* 33, 1 (1986), 151–178.
[16] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* (2016), 1–27.
[17] Susanne Graf and Hassen Saïdi. 1997. Construction of abstract state graphs with PVS. In *International Conference on Computer Aided Verification*. Springer, 72–83.
[18] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.
[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches.. In *USENIX Security Symposium*. 897–912.
[20] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache storage channels: Alias-driven attacks and verified countermeasures. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 38–55.
[21] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–Bringing access-based cache attacks on AES to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 490–505.
[22] Zecheng He and Ruby B Lee. 2017. How secure is your cache against side-channel attacks?. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 341–353.
[23] Saul Kripke. 2007. Semantical considerations of the modal logic. (2007).
[24] Leslie Lamport. 1980. Sometime is sometimes not never: On the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 174–185.
[25] Kim G Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *International journal on software tools for technology transfer* 1, 1-2 (1997), 134–152.
[26] Ruby B Lee, Peter Kwan, John P McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for protecting critical secrets in microprocessors. In *ACM SIGARCH Computer Architecture News*, Vol. 33. IEEE Computer Society, 2–13.
[27] Fangfei Liu and Ruby B Lee. 2014. Random fill cache architecture. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 203–215.
[28] Yangdi Lyu and Prabhat Mishra. 2017. A Survey of Side-Channel Attacks on Caches and Countermeasures. *Journal of Hardware and Systems Security* (2017), 1–18.
[29] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*. Springer, 1–20.
[30] Colin Percival. 2005. Cache missing for fun and profit.
[31] Amir Pnueli. 1977. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE, 46–57.
[32] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23, 1 (2010), 37–71.
[33] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2016. SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.
[34] Zhenghong Wang and Ruby B Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 494–505.
[35] Zhenghong Wang and Ruby B Lee. 2008. A novel cache architecture with enhanced performance and security. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*. IEEE, 83–93.
[36] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 347–360.
[37] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are Coherence Protocol States Vulnerable to Information Leakage?. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 168–179.
[38] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*. 719–732.
[39] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. 2012. Language-based control and mitigation of timing channels. *ACM SIGPLAN Notices* 47, 6 (2012), 99–110.
[40] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A hardware design language for timing-sensitive information-flow security. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 503–516.
[41] Tianwei Zhang and Ruby B Lee. 2014. New models of cache architectures characterizing information leakage from cache side channels. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 96–105.
[42] YongBin Zhou and DengGuo Feng. 2005. Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. *IACR Cryptology ePrint Archive* 2005 (2005), 388.