Analyzing The Security of The Cache Side Channel Defences With Attack Graphs

Limin Wang[1,2] Ziyuan Zhu[1,2,(✉)] Zhanpeng Wang[1,2] Dan Meng[1]

[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China
e-mail:{wanglimin, zhuziyuan, wangzhanpeng, mengdan}@iie.ac.cn

Abstract— Note that very limited work is proposed to analyze the security of defenses against the cache side channel attacks on micro-architecture. In this paper, we propose a model based method to generate a visual attack graph and analyze the security of micro-architecture security designs in the early stages of processor design. The experiments indicate that our method can identify the special attack paths that some common security designs fail to defend against and show them in an attack graph.

I. Introduction

Cache based side channel attacks like flush+reload, evict+reload, prime+probe, and evict+time are able to leak the secret data due to the difference of access time between cache hits and misses. Fortunately, various of traditional software and hardware methods can mitigate side channel vulnerabilities well [1]. However, since it was reported in 2018, the hardware vulnerabilities like Meltdown and Spectre have been attracting a lot of interest [2], [3]. These hardware vulnerabilities can diversify attack methods in some steps of the cache attacks. For example, in the flush+reload cache attack. Previously, the attacker has to wait for the victim to load secret data into the cache, but now, the victim's secret data can be loaded in the cache by Spectre or Meltdown which exploits speculative execution vulnerabilities. With these vulnerabilities that exploit on the micro-architecture features, cache side channel attacks are more powerful. As a result, two new problems arise: (1) Whether the traditional cache-only protections or the micro-architecture countermeasures can also provide high-quality defense against the new cache side channel attacks that exploit the hardware vulnerabilities? (2) If these security designs are not secure enough, how did they fail?

Unlike software, hardware problems are notoriously difficult to solve after they are published. If we can answer the questions above, we will help engineers know the remaining vulnerabilities in the processor with countermeasures when they are designing, so that they can fix it in the early stages of processor design, which can greatly improve the security of the hardware with low cost.

In order to answer the two questions above, some work has been done to help evaluate cache side channel security. Some methods have been proposed to analyze the security of the secure cache architecture through computer simulation experiments [4], [5], and other approaches use abstract model analysis to help evaluate the secure cache architecture [6], [7], [8]. However, they can not analyze the security of the security designs on other micro-architecture components that can systematically defend against the new cache side channel attacks. To solve the requirements for analyzing the security of micro-architecture countermeasures, Alloy analyzer and "micro-architecturally happens-before"(μhb) graphs are introduced in [9]. However, the μhb graphs used in such an approach eventually lead to poor readability, which makes it difficult for engineers to locate the defects in security designs.

In this paper, to answer the question (1), we use instruction abstract method to model the micro-architecture and its security designs, which can be described by NuSMV language as a Kripke structure, and we use computation tree logic (CTL) to express the security specification that the model should never reach the insecure states caused by the attacker's successful attack. With the model and the CTL formula, the modified model checker NuSMV will try to search the whole state space of the model to find the possible attack paths that violate the security specifications [10]. If no attack path can be found, the security designs on micro-architecture can be considered to be able to protect the data from the specified cache side channel attack. As for question (2), if the modified NuSMV can find attack paths, the attack graph generated can visualize the attack paths and show the different vulnerabilities exploited by different attack paths, so that it can help users to find out the special attack cases that are not considered in security designs.

To validate the method proposed in this paper, we will verify and analyze some secure cache architectures such as Static Partitioning Cache(SP Cache) and secure speculation designs like InvisiSpec [11], [12], [6], and we successfully prove that SP Cache can resist several side channel attacks, however, flush (evict)+reload with Spectre attack can still bypass it.

In this paper, the key contributions include:

1. The paper proposed a new model based cache side channel defense security analysis method which can analyze the security of the micro-architecture security designs.
2. We use the sequence of crucial states rather than the pattern of the attack methods to express the security specification, so that we can enumerate the known and unknown attack paths that are able to reach these crucial states.
3. To conveniently analyze the deficiencies in countermeasures, our method extends the attack graphs and proposes a novel use of the attack graph technologies to visualize the cache side-channel attack paths.

II. METHODOLOGY

The early-stage security analysis method proposed in this paper can be divided into two steps: *Attack Data Generation* and *Attack Graph Generation*. Both steps are described in detail after an overview.

978-1-7281-4123-7/20/$31.00

A. Overview

Fig.1 presents an overview of the side channel defense security analysis method. In this tool flow, the micro-architecture model described by NuSMV language and the security specification described by CTL are the inputs of the modified NuSMV. Then the modified NuSMV will verify whether the model satisfies the specification, if the model satisfies, the modified NuSMV will output the result *OK*, otherwise, the modified NuSMV will generate multiple counterexamples to prove the security designs are not secure enough. Then an attack graph will be generated based on these counterexamples. In order to make the final attack graph the tool flow outputs reduced and readable, the graph reduction module is necessary.

B. Attack Data Generation

1) Micro-architecture Model: In this paper, we will use the instruction abstract method to model micro-architecture. The method executes an abstract instruction per step in program order, and the executing instruction will lead to state transitions of the micro-architecture [13], [14], [15].

The micro-architecture model is a Kripke structure that can express the state transitions, and it will be described by NuSMV language.

Definition 1 Kripke structure M is a 4-tuple, let $M = (S, I, R, L)$, and AP be a set of atomic propositions.

- S is a set of states. Every micro-architecture state in S usually contains the states of the several properties. For example, the properties of the different micro-architecture components and the properties of an attacker program and a victim program.
- I is a set of initial states.
- R is a transition relation, $R \subseteq S \times S$. It includes the state transitions triggered by the abstract instructions or the constraints between micro-architecture components.
- L is an interpretation function that maps from each state to the set of atomic propositions that are true in that state, $L : S \longrightarrow 2^{AP}$.

The necessary properties and the state transitions in model M will be illuminated detailedly in the following.

Micro-architecture Components. The micro-architecture components and their properties we need should be chosen at first. For instance, to verify whether a secure cache architecture design is secure under the cache side channel attacks, cache is a necessary component and some of its properties like whether the victim's secret data is in the cache, denoted as *ExistSC (abbreviated as sc)* and whether the attacker's general data is in the cache, denoted as *ExistGN (abbreviated as gn)* should be added into each state in S.

The more components and properties are modeled, the more precise the model verification is. However, if too many properties are added into the model, the state space will explode during the model checking. Sensibly selecting components and properties to be modeled plays an important role in allowing the engineers to make a tradeoff between the precision and the complexity.

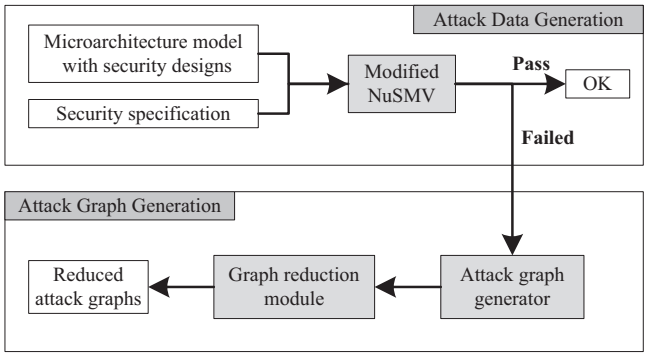

Fig. 1. Overview of cache side channel defense security analysis method. The large rectangle above represents the first step to generate attack data, and the large rectangle below indicates the second step to generate attack graphs. The little gray rectangles inside are tools, the arrows before and after tools mean inputs and outputs.

Abstract Instructions. The abstract instruction set will also help to decrease the complexity of the model. *load*, *store*, *jump*, *branch* are frequently-used abstract instructions, and some special but important instructions like *clflush* can also be included in the abstract instruction set as needed. *clflush* is an x86 instruction to invalidate the cache line that contains the specified linear address in all levels of the cache, it is often used in cache side channel attacks.

The state of the model M changes by executing the abstract instructions, denote the abstract instructions set as A, and denote the conditions that have to been satisfied to execute the instructions as F, and the semantics of execution of an abstract instruction is as follows.

$$s \xrightarrow{a(a\in A)\ ,\ f_a\subseteq s(f_a\in F, s\in S, s'\in S)} s' \tag{1}$$

Formula (1) shows that only when the instruction a runs and current state s satisfies what a needed, can state s change to s'. All of the state transitions like Formula (1) should be added into the set R.

In addition to the state transitions directly triggered by the abstract instructions, there are some state transitions triggered by the property changes in another micro-architecture component, and these constraints between different components should also be modeled. Take the cache coherence protocol for example, assume there are 2 cores, and their caches have the same data. When the data in one core is modified, the data in another core will become invalid. The state transitions triggered by MESI protocol should be added into R.

Attacker and Victim. The attacker and victim programs are a sequence of abstract instructions. Every abstract instruction in the programs will affect the current state of the micro-architecture. Once the state becomes a dangerous state, it means the attack succeeds.

For example, the current abstract instruction the attacker and the victim is executing can be denoted as *AttackerOP (aop)* and *VictimOP (vop)*. They are properties of an attacker and a victim, and both of them should be added into every micro-architecture state in S. At the moment, if *AttackerOP* is *store*, the secret data is in the cache, and the attacker aims to occupy the cache sets where the victim's secret data is stored, so that the secret data will be evicted out of the cache. Then the property *ExistSC* will change from *true* to *false*.

The secure problem here is the external interference between the attacker and the victim [7]. Specifically in this example, what the attacker does can affect the victim's data in cache. Some security designs like SP Cache are used to solve this problem. SP Cache isolates the victim and the attacker to ensure that they do not share partitions, after applying the SP Cache to our processor, the state transitions triggered by *store* mentioned above will not be in R.

2) Security Specification: In this paper, the aim of the security designs is to ensure the safety property that the system should never reach the *dangerous* states, and the safety property can be described as a security specification in CTL, which is shown in (2). The CTL formula symbols can be found in Table I.

$$\neg EF(dangerous) \tag{2}$$

For cache side channel attacks, even though exploits are increasing rapidly, but the relevant insecurity states do not. Therefore, we can conveniently develop security specifications for them. In different exploits, the attacker uses different vulnerabilities to make the system reach the same insecure state. These states that have to be satisfied during the attack are called crucial states in our paper. A cache side channel attack can be divided into several attack steps, and when the current crucial state meets what the attack step needs, the attack step can be triggered. If this attack step is successful, then it will make the system become another crucial state the attackers expected. If the current state meets what the next attack step needs, the subsequent attacks will continue. As long as one of these crucial states is not satisfied, the attack fails. For example, in the flush+reload attack, if the computer system has the instruction *clflush*, the first attack step that flushing the cache can be triggered, if successful, the specified cache data will be flushed, and then the next attack step can be performed. In the first step of the evict+reload attack, the attackers can also reach the same crucial state by evicting the cache.

The *dangerous* in formula (2) can be described as a sequence of crucial states in CTL, and then is used in the modified NuSMV. In this way, the modified NuSMV can find the known and even unknown attack paths that can reach the crucial states in *dangerous* as much as possible.

3) Model Checking In The Modified NuSMV: In CTL model checking, given a Kripke structure M, a computation path of M is a sequence of state transitions start from one of the initial states I, and the state transition rules can be found in R. With transition relation R and initial states I, M can represent all of the possible paths from the initial states. To verify whether the model satisfies the specification (2), the modified NuSMV will try to check all of the computation paths, and find a path that satisfies the *dangerous*. If the modified NuSMV finds it, this path will be shown as a counterexample. According to the counterexample, we can know every step that how the model change from initial normal states to reachable crucial states and what the attacker and the victim do in each step. Denote that the modified NuSMV in our paper can find multiple counterexamples, and we can set an upper bound to limit the search length of a counterexample as required. Every counterexample found is an attack path, and all of the attack paths can be used to generate an attack graph, which is readable and very convenient to help users analyze the vulnerabilities that are here to stay.

TABLE I
SYMBOLS OF CTL OPERATORS

Operations[a]	Symbols	Description
Path Quantifier	A	for *A*ll of the path
	E	there *E*xists at least one path
Temporal Symbols	Xϕ	ϕ holds at ne*X*t time
	Gϕ	ϕ holds *G*lobally,
	Fϕ	ϕ holds at *F*uture
	ϕUφ	ϕ holds *U*ntil φ holds

[a] In CTL, every CTL operation is always a combination of the path quantifier and the temporal symbol. For example, $EF\phi$ denotes there exists at least one path that ϕ holds at future.

C. Attack Graph Generation

1) Attack Graphs: The formal model checking method is rigorous, so that there are many redundant paths in the counterexamples the modified NuSMV generates. To help users analyze the attack paths easily, attack graphs are utilized to simplify and visualize these attack paths.

Algorithm 1 shows how to turn multiple counterexamples into an attack graph. An attack graph consists of several attack paths, and an attack path includes nodes and edges. The nodes represent the states of micro-architecture, the edges between nodes are the abstract instructions the attacker and the victim are executing. This algorithm will traverse all of the counterexamples, and determine whether the state in the counterexample is a crucial state by comparing with the crucial states in security specification, and then record the results in the *AGNode*. Finally, the counterexamples will be translated to the attack paths, and then are rendered to an attack graph.

2) Attack Graph Reduction : Fig.2 shows the three situations that influence the readability of attack graphs. In Fig.2, (a), most states in the two attack paths are the same, this situation will be solved by merging the same states when generating the attack graph, the merged attack paths are presented in Fig.2, (b). When there are a number of attack paths in an attack graph, it is beneficial to merge the same states so that the security problems become easier to find for engineers. Fig.2, (c) represents two repeated attack paths and Fig.2, (d) shows two logically equivalent attack paths, if the attack path $s1 \rightarrow s2 \rightarrow s3 \rightarrow s4$ means a successful attack, the path $s1' \rightarrow s2' \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow s4$ can also represent a successful attack, so the longer one is redundant. Fig.2, (c) and Fig.2, (d) will be reduced in the graph reduction module.

III. CASE STUDY: CACHE SIDE CHANNEL DEFENSE SECURITY ANALYSIS

In this section, we use a simple case to show how to analyze the security of the security designs under the cache side channel attacks flush+reload with Spectre attack and evict+reload with Spectre attack.

Algorithm 1 Generate An Attack Graph

Input:

Counterexamples *ce*: every counterexample *ce*[*i*] is a list of states.

crucial states and their names: *i_states* and *i_names*.

Output:

an attack graph *AG*: *AG* is a list of *AP**.

```
1:  typedef struct AGNode {
2:        DataType   state, aop, vop;
3:        BOOL       is_crucial_state;
4:        STRING     name_of_state
5:  } AGNode, AP*;
6:
7:  for each counterexample ce[i] do
8:     AP* ap; /* ap is an attack path */
9:     for each state ce[i][j] do
10:       AGNode node;
11:       node.aop = ce[i][j].aop;
12:       node.vop = ce[i][j].vop;
13:       remove aop and vop from ce[i][j];
14:       node.state = ce[i][j];
15:       if node.state ∈ i_states then
16:          node.is_crucial_state = TRUE;
17:          node.name_of_state = find the name of node.state
             in i_names;
18:       end if
19:       add node into ap;
20:    end for
21:    add an attack path ap into AG;
22: end for
23: AG = reduceAttackGraph(AG);
24: return AG;
```

Micro-architecture Model. Note that to make the case easy to understand, this micro-architecture model will be as simple as possible. For example, multi-cores and pipelines are not necessary for flush (evict)+reload with Spectre, so they are not added into the model, but they can also be modeled if needed [15], [16].

Assume the micro-architecture model in this case is *M*, *M* is a Kripke Structures. Table II shows all of the Micro-architecture properties included in S. *ExistSC*, *ExistGN*, *AttackerOP* and *VictimOP* have been explained in Section II. *PredictionResult* represents whether the result of the branch predictor is correct, *TSuccessful* and *TFailed* mean both of the outcomes of the branch predictor are *Taken*, the predicted results are correct and wrong respectively. On the contrary, *NTSuccessful* and *NTFailed* represent the outcomes are *Not Taken*. *Mode* shows the operation that the processor is performing, for example, after the branch predictor fails to predict the result of a *branch*, the processor will have to squash the pre-executed instructions, then *Mode* will be labeled as "squash". *RWAddr* means the address an attacker or a victim accesses. *Wtime* is a little special, it is a flag that records the times of an attacker continuously performs write operations, when *Wtime* is larger than *n* (*n*=3 in this paper), that means the attacker has evicted the victim's data out of the cache.

Then we will use NuSMV language to describe the transition relation *R*. According to the formula (1), *R* can be obtained in Fig.3, which shows even when executing the same abstract instructions, if the current states are different, the transitions will be different.

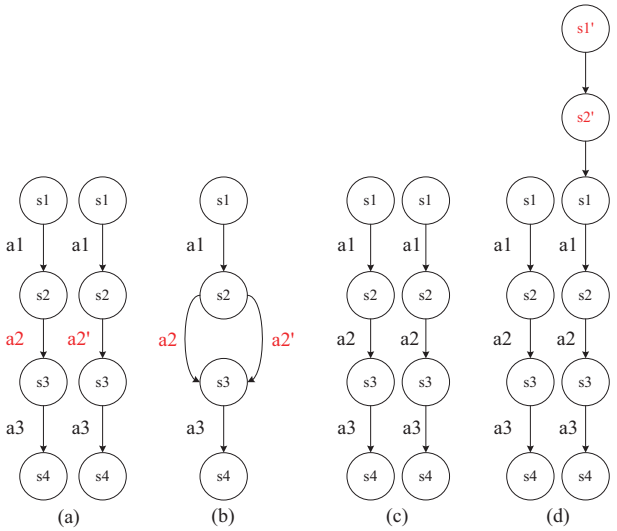

Fig. 2. (a) Attack paths with similar structure. (b) Attack paths that merging the similar structure (c) Attack paths with the same structure. (d) Attack paths with useless structures.

Security Specification. As Fig.4 shows, the flush+reload attack can be divided into three steps. At first, the attacker has to clean a cached memory location, the memory address is carefully selected. Then the attacker wait for the victim to load the secret data to the cache position that was just flushed. Finally, the attacker will try to access the memory location again, the fast access speed means the victim has loaded the data, and the data was cached in that flushed position. The evict+reload attack is similar to the flush+reload, however, instead of using *clflush*, the attacker evicts the victim's data by continuously accessing the memory until the secret data in the cache is replaced and no longer cached.

The Spectre attack can train and mislead the branch predictor to make a wrong speculative prediction. When speculative execution, the executing instructions of an attacker can access the out-of-bounds address, then the data information accessed will be cached. However, when the branch predictor finds the incorrect prediction and squashes the speculated instructions, the cached secret data will not be cleaned.

With Spectre attack, at the second step of flush (evict)+reload, the attacker does not have to wait for the victim, and they can exploit Spectre to access out-of-bounds memory address where the victim's data is stored, then the secret data information will be also cached.

According to Section II, to check and analyze the vulnerability of the security designs in micro-architecture, we need to define security specifications based on the crucial states in Fig.4. The crucial state *S2* means that after flushing or evicting, the secret data is not in the cache (*sc=false*), and *S3* represents that after Spectre attack (*md=squash*), the secret data information was cached in that flushed or evicted position again(*sc=true*). To block the attack paths in Fig.4, *S2* or *S3* should not be satisfied, which is the aim of the security designs on micro-architecture.

TABLE II
SELECTED MICRO-ARCHITECTURE COMPONENTS AND PROPERTIES

P_i[a]	Components	Properties (abbr.)		Value
P_1	Cache	ExistSC	(sc)	boolean
		ExistGN	(gn)	boolean
P_2	Branch Predictor	Prediction-Result	(pr)	TSuccessful, TFailed NTSuccessful, NTFailed
P_3	Processor	Mode	(md)	normal, squash prediction, evict
P_4	Attacker	AttackerOP	(aop)	clflush, load, store, branch
		RWAddr	(addr)	addr_sc, addr_gn
P_5	Victim	VictimOP	(vop)	clflush, load, store, branch
		RWAddr	(addr)	addr_sc, addr_gn
P_6	Flag	Wtime	(tm)	unsigned integer

[a] $P = (p \mid p \in \bigcup_{i=1}^{6} P_i)$, P_i is a set of *Properties* in Row i. Every micro-architecture state in S should contains the properties in P.

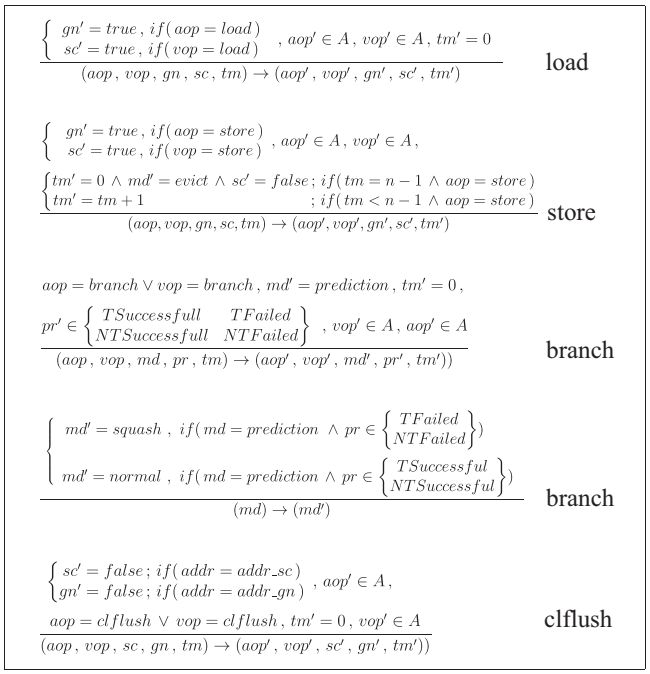

Fig. 3. The transition relation built based on the formula 1 for abstract instruction *load*, *store*, *branch*, and *clflush*. The *branch* has 2 transitions, the first one is triggered by abstract instruction, the second one is triggered by the outcomes predicted by branch predictor.

Safety properties 1 *What the attacker does can never affect the victim's data in the cache.*

Safety properties 2 *Cache should also be cleaned when squashing.*

According to the *S2* and *S3*, we can summarize the *Safety properties 1* and *Safety properties 2*. If the micro-architecture model satisfies *Safety properties 1* and *Safety properties 2*, the security designs will be considered to be secure enough under the flush (evict)+reload attacks with Spectre. These safety properties can be manually described as a security specification in CTL, which is shown in formula (3).

$$\neg EF(E[sc = false\ U((md = squash)EX(sc = true))]) \quad (3)$$

The *dangerous* state in formula (3) means that there is a path, and in this path, one of the states (sc = *false*) represents that the first step of flush + reload is successful, and the data of the victim in the cache is flushed by an attacker. Then the state holds until NuSMV finds another state (*md*=*squash*), this state represents that at the second step, Spectre attack is successful, the secret data of the victim has been cached, and the processor is squashing after the wrong prediction caused by the attacker. And then NuSMV finds that in the next state (sc=true), the secret data cached during the Spectre attack is still in the cache after squashing. Formula (3) means that if there exists no path that can match *dangerous*, the micro-architecture with secure designs will be considered secure enough under flush+reload with Spectre.

Experiments. With the model and specification above, the attack graph will be generated. Fig.5, (a) is the attack graph of micro-architecture without any security designs, the circles represent the states of micro-architecture, and the arrows between two circles represent the abstract instructions, the red rectangles represent the crucial states. It is clear that there are many attack paths, which means the micro-architectures without any security designs are not secure under the flush (evict)+reload with Spectre attack.

Then we use SP Cache to try to block the attack paths. With this countermeasure, evicting others' data out of the cache by continuously *store* will be not allowed. The new attack graph generated is shown in Fig.5, (b).

Fig.5, (b) and Table III shows the model with SP Cache still have 81 counterexamples. By contrast with Fig.5, (a), we find the SP cache can successfully defend against the evict+reload attack, however, the micro-architecture is still insecure under the flush+reload attack.

In addition, the two leftmost attack paths are remarkable unknown variants. When the attack starts, if the victim's secret data have not been stored in the cache partition of the victim yet, the first attack step will be not necessary (the final accuracy will be lower). And in the second step, due to the Spectre attack, the attacker can access the victim's address illegally, then the data accessed can be mapped to the attacker's address, which is cached to attacker's cache partition. And in the third step, the attacker can probe the secret data as usual in his cache partition. The cases shows that the recently discovered Spectre attack will help the traditional flush (evict)+reload attacks bypass the protection of the SP cache. In general, The SP Cache works, but it can only make the *S2* caused by *evict* unsatisfied.

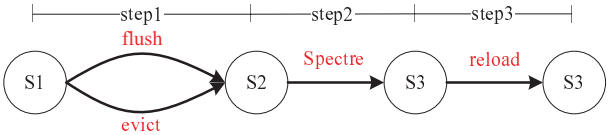

Fig. 4. The attack steps in flush+reload attack and evict+reload attack.

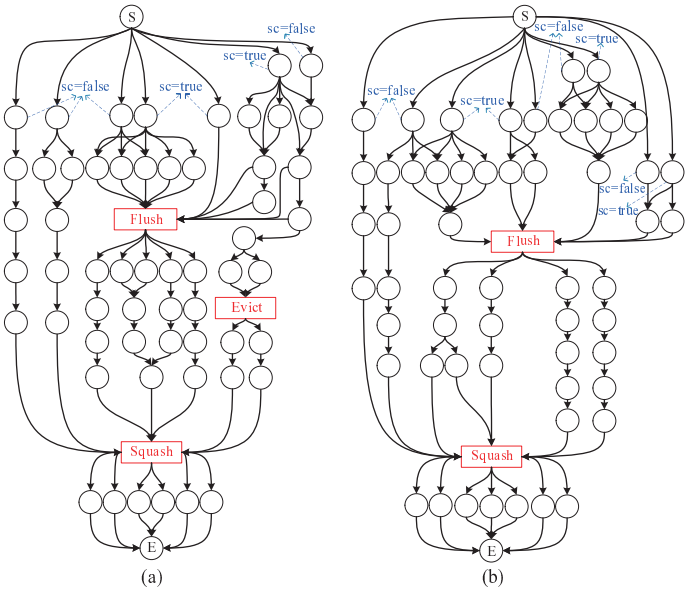

Fig. 5. (a) is the attack graph of micro-architecture without any security designs, and (b) is the attack graph of micro-architecture with SP Cache

In order to compensate for this deficiency, the alternative countermeasure is the combination of SP cache and InvisiSpec, InvisiSpec load data into a new Speculative Buffer instead of cache until the speculative load is finally safe. The aim of the InvisiSpec is to help make the *S3* unsatisfied. According to Fig.4, if the Spectre attack can not reach the crucial state *S3*, the attack paths of flush (evict)+reload attacks will be blocked successfully. And Table III shows that there is really no counterexample. The result indicates that the combined security design can defeat the flush (evict)+reload with Spectre attack.

We can build new security specifications based on different attacks, then analyze and improve the security of the security designs until they met our security needs.

IV. Summary and Conclusions

This paper proposed a new method that using model checking method to verify whether the security designs of micro-architecture are secure enough under the side channel attacks and using attack graphs to analyze where the security designs can not protect. The method has the advantages as follows: (1) It is able to use instruction abstract method to conveniently model the micro-architecture as a Kripke structure. (2) It can find some unknown attack paths by building and verifying the security specification based on a sequence of crucial states. (3) It can remove and combine the redundant attack paths, and visualize the attack paths with the attack graph technologies.

However, there are still some limitations, for instance, we still can not solve the state space explosion problem, which is a challenge in model checking. In addition, our method can only check whether the security designs satisfy the security specifications, which means the attacks do not violate the safety properties described by security specifications will not be identified.

In the future, we plan to quantify the attack graph, which will help users to identify and defend against the high-risk attack paths with less effort and cost. In addition, we will explore modeling automation to decrease the workload of users.

TABLE III
Security Verification Result Of Micro-architecture With Different Security Designs

Secure Designs	Bounded	Counterexamples (Number)	Reduced Attack Paths (Number)	Runtime (s)
None	9	247	22	4.421
SP Cache	9	81	20	3.404
SP Cache InvisiSpec	9	0	0	0.849

References

[1] F. Zhang, Z. Y. Liang, B. L. Yang, X. J. Zhao, S. Z. Guo, K. Ren, "Survey of design and security evaluation of authenticated encryption algorithms in the CAESAR competition," *Frontiers of Information Technology & Electronic Engineering*, vol. 19, pp. 1475–1499, 2018.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, et al, "Meltdown: reading kernel memory from user space," *27th USENIX Security Symposium (USENIX Security 18)*, pp. 973–990, 2018.

[3] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, et al, "Spectre attacks: exploiting speculative execution," *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019.

[4] T. Zhang, F. Liu, S. Chen, R. B. Lee, "Side channel vulnerability metrics: the promise and the pitfalls," *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ACM, 2013.

[5] J. Demme, R. Martin, A. Waksman, S. Sethumadhavan, "Side-channel vulnerability factor: a metric for measuring information leakage," *39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 106-117, IEEE, 2012.

[6] Z. He, R. B. Lee, "How secure is your cache against side-channel attacks?," *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2017.

[7] T. Zhang, R. B Lee, "Secure cache modeling for measuring side-channel leakage," *Technical Report, Princeton University*, 2014.

[8] T. Zhang, Y. Zhang, R. B. Lee, "Analyzing cache side channels using deep neural networks," *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 174-186, ACM, 2018.

[9] C. Trippel, D. Lustig, M. Martonosi, "CheckMate: automated synthesis of hardware exploits and security litmus tests," *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 947-960, IEEE, 2018.

[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, et al, "Nusmv 2: an opensource tool for symbolic model checking," *International Conference on Computer Aided Verification*, pp. 359-364, Springer, 2002.

[11] Z. Wang, R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *ACM SIGARCH Computer Architecture News*, pp. 494-505, 2007.

[12] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, J. Torrellas, "InvisiSpec: making speculative execution invisible in the cache hierarchy," *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 428-441, IEEE, 2018.

[13] S. K. Lahiri, R. E. Bryant, "Deductive verification of advanced out-of-order microprocessors," *International Conference on Computer Aided Verification*, pp. 341–354, Springer, 2003.

[14] R. E. Bryant, "Term-level verification of a pipelined CISC microprocessor," 2005.

[15] R. Jhala, K. L. McMillan, "Microarchitecture verification by compositional model checking," *International Conference on Computer Aided Verification*, pp. 396-410, Springer, 2001.

[16] Y. Gao, X. Li, "Formal verification of out-of-order processor," *International Conference on Computer Modeling and Simulation*, pp. 129-135, IEEE, 2009.