A Security Verification Template to Assess Cache Architecture Vulnerabilities

Ghasempouri, Tara; Raik, Jaan; Paul, Kolin; Reinbrecht, Cezar; Hamdioui, Said; Taouil, Mottaqiallah

DOI
10.1109/DDECS50862.2020.9095707

Publication date
2020

Document Version
Accepted author manuscript

Published in
2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)

A Security Verification Template to Assess Cache Architecture Vulnerabilities

Tara Ghasempouri, Jaan Raik, Kolin Paul
Tallinn University of Technology
{tara.ghasempouri, jaan.raik, kolin.paul}@ttu.ee

Cezar Reinbrecht, Said Hamdioui, Mottaqiallah Taouil
Delft Univeristy of Technology
{C.R.WedigReinbrecht, S.Hamdioui, M.Taouil}@tudelft.nl

***Abstract*—In the recent years, cache based side-channel attacks have become a serious threat for computers. To face this issue, researches have been looking at verifying the security policies. However, these approaches are limited to manual security verification and they typically work for a small subset of the attacks. Hence, an effective verification environment to automatically verify the cache security for all side-channel attacks is still missing. To address this shortcoming, we propose a security verification methodology that formally verifies cache designs against cache side-channel vulnerabilities. Results show that this verification template is a straightforward, automated method in verifying cache invulnerability.**

I. Introduction

Today, software attacks can compromise hardware security through so-called Logical Side-Channel Attacks (LSCAs) [1]. Adversaries can use LSCAs to understand the system's secrets by simply observing its behavior, e.g. the timing required to access data from the cache memory. Several examples of cache attacks have been reported in recent years [2–7]. Such attacks represent a serious threat for the semiconductor industry. Patching vulnerabilities in the field may be very costly, degrade the performance, and sometimes even creates new issues (e.g. reduced battery lifetime of an IoT device) [8]. Therefore, there is a strong need for identifying security vulnerabilities using verification methods during design time, while considering both the architecture and threat models.

Several articles address the topic of security verification [9–12]. In [9], Jha et al. presented a verification scheme that targeted the validation of access policies in hardware by different agents. This work is limited to the evaluation of security policies only and does not consider real threats and vulnerabilities. In [10], Subramanyan et al. proposed a verification methodology to evaluate the propagation of system vulnerabilities using taint-properties (i.e. properties related to information flow and access control) in a design. Some predefined flows of information are classified up-front as insecure and the verification methodology verifies in an automated fashion the occurrence possibility of such a flow in the design. This technique detects security flaws, however, LSCAs cannot be detected as even the secure flows leak information. Regarding this aspect, Deng et al. presented in [11] and [12] novel approaches to model cache attacks. In both papers, the authors modelled some famous attacks and thereafter manually verified the security of different cache solutions. However, they did not present a methodology or an *automated* way of evaluating the cache designs under attacks. The above clearly shows that an automated security verification method which includes the design, information flow, as well as the threat and attack models is missing.

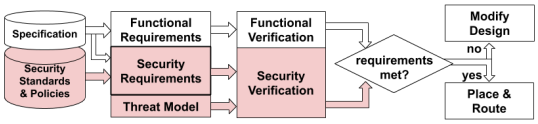

Fig. 1: Functional versus security verification flow

To address this shortcoming, we propose a formal security verification template to assess the security of a hardware design in an *automated* way during the early design phase. Fig. 1 shows the design flow where the security elements have been integrated in red. Traditionally, a design starts by converting a design specification into functional requirements which are subsequently translated into a design. During this process, the design is translated from functional requirements to a netlist [13]. To guarantee that the design is correctly translated during this process, functional verification is applied at every transition between the abstraction levels [14]. In a similar manner this concept can be extended to the verification of the security requirements. We demonstrate the correctness of the proposed security verification methodology by analyzing the vulnerabilities related to side-channel analysis of nine cache designs presented in the literature, i.e., one conventional non-partitioned cache [15], three statically-partitioned caches [16–18], and five dynamically-partitioned caches [19–23]. The main contributions of this paper are:

- An innovative automated verification template to evaluate the security of a cache design, i.e., whether it is vulnerable to predefined attacks or not.
- A method to analyze the attack models
- An algorithm that automatically synthesizes attack models into a table with attacks.
- A demonstration of the proposed methodology for the nine cache designs by analyzing their security requirements using an exhaustive set of 28 attack models (LSCA) [11].

The remainder of this paper is organized as follows. Section II provides the required background information. Section III presents our proposed verification methodology. Section IV shows the experimental results. Section V presents a brief discussion. Finally, Section VI concludes this paper.

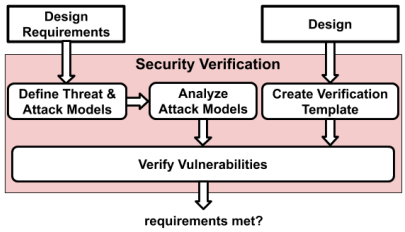

Fig. 2: The proposed Security Verification Methodology

II. Background on Cache Attacks

Most cache attacks take advantage of the fact that cache hits and misses are dependent on the cryptographic key [24]. When a cache miss occurs, the cache coherency mechanism initiates an access to a higher memory level and hence needs a longer time to fetch the data. Therefore, by observing traces with cache misses and hits, information regarding the value of the secret key can be retrieved [24]. The first attack proposals for caches targeted the secret key of different cipher implementations, such as DES, RSA or AES [2, 3, 25]. Based on how the leakage information is collected, cache attacks can be classified in access-based, timing-based and trace-based cache attacks [24].

Access-based cache attacks analyze which addresses were accessed during a cryptographic cipher operation by measuring the time the attacker needs to access his own data [3, 26]. Timing-based cache attacks measure the total execution time of a cryptographic cipher operation (encryption or decryption). As the number of cache misses and hits are key dependent, different keys lead to a different execution time [2, 27]. Finally, trace-based attacks collect information regarding cache hits and misses using external monitors [28, 29]. These attacks are out of the scope of this paper as they are not cache exclusive attacks, i.e., they exploit also other components in the system, such as High Performance Counters (HPC).

III. Security Verification Methodology

Figure 2 presents the main steps of the proposed verification platform. The inputs of the framework are the design model and the design requirements. The output is a verified cache model and a report describing whether the design is secure against the different cache attacks or not. As shown in Figure 2, the framework is divided into four main steps: *A) Define Threat and Attack Models*, *B) Analyze Attack Models*, *C) Create Verification Template*, and *D) Verify Vulnerabilities*. Next, we describe each step in more detail.

A. Define Threat and Attack Models

The threat model defines the main vulnerabilities of the system. In other words, the threat model is composed of a set of realistic assumptions and definitions of what an adversary can and cannot do in the system. In addition, the threat model can be accompanied with attack models. An attack model describes the behavior of a practical attack.

Our methodology adopts the exhaustive set of 28 cache attack types proposed by Deng et al. in [11]. These attack types are described in Table I, where each type is referenced by a specific ID, i.e., from 1 to 28. The attack types are described by a $formula$ containing three symbols with two arrows between them. Next we describe the definitions and the related assumptions (in form of axioms) that composes our threat and attack model.

TABLE I: Attack Database: 28 types analyzed in this work

ID	attack $formula$	*	ID	attack $formula$	*
1	$Vx \rightarrow Ar \rightarrow Vx$	-	15	$Vx \rightarrow Vx \rightarrow Ar$	d
2	$Vx \rightarrow Vr \rightarrow Vx$	-	16	$Ar \rightarrow Vx \rightarrow Vr$	d
3	$Ar \rightarrow A1 \rightarrow Vx$	-	17	$Vr \rightarrow Vx \rightarrow Vr$	d
4	$Vr \rightarrow A1 \rightarrow Vx$	-	18	$Vx \rightarrow Vx \rightarrow Vr$	d
5	$A1 \rightarrow A1 \rightarrow Vx$	-	19	$Ar \rightarrow Vx \rightarrow A1$	e
6	$V1 \rightarrow A1 \rightarrow Vx$	-	20	$Vr \rightarrow Vx \rightarrow A1$	e
7	$Vx \rightarrow A1 \rightarrow Vx$	a	21	$A1 \rightarrow Vx \rightarrow A1$	f
8	$Vx \rightarrow A1 \rightarrow Vx$	b	22	$V1 \rightarrow Vx \rightarrow A1$	-
9	$Vr \rightarrow V1 \rightarrow Vx$	b	23	$Vx \rightarrow Vx \rightarrow A1$	e
10	$A1 \rightarrow V1 \rightarrow Vx$	b	24	$Ar \rightarrow Vx \rightarrow V1$	b
11	$V1 \rightarrow V1 \rightarrow Vx$	b	25	$Vr \rightarrow Vx \rightarrow V1$	b
12	$Vx \rightarrow V1 \rightarrow Vx$	c	26	$A1 \rightarrow Vx \rightarrow V1$	-
13	$Ar \rightarrow Vx \rightarrow Ar$	d	27	$V1 \rightarrow Vx \rightarrow V1$	c
14	$Vr \rightarrow Vx \rightarrow Ar$	d	28	$Vx \rightarrow Vx \rightarrow V1$	b

*Published attacks: a) Evict+Time [3]; b) Cache Collision [24]; c) Berstein [2]; d) Flush+Flush [27]; e) Flush+Reload [26]; f) Prime+Probe [3]

Definition 1. ***V1*** *is a specific cache address that is accessed by the victim which is known to the attacker.*

Definition 2. ***Vx*** *represents an access by the victim using a specific cache address which is not known to the attacker.*

Definition 3. ***Vr*** *is a flush operation initiated by the victim for one or more addresses which is/are not known to the attacker.*

Definition 4. ***A1*** *represents an access by the attacker using a specific cache address.*

Definition 5. ***Ar*** *is a flush operation initiated by the attacker for one or more addresses.*

Definition 6. *An attack $formula$ is a sequence of three operations, given by:*

$$attack \Leftrightarrow \{\ Stage\ 1 \rightarrow Stage\ 2 \rightarrow Stage\ 3\ \}$$

Definition 7. ***Stage 1 (i.e, setup phase)*** *is the step where the attack is prepared. Usually, a memory operation performed by either the victim or attacker initializes a single cache line to a certain initial state.*

Definition 8. ***Stage 2 (i.e., trigger phase)*** *is the step where the initialization data is modified. This can be done by the victim or attacker.*

Definition 9. ***Stage 3 (i.e., observation stage)*** *is the step where the attacker gathers information. For example, the time required to complete the memory access (by the attacker or victim) is observed by the attacker during this stage.*

Axiom 1. *When a victim (Def. 1, Def. 2, and Def. 3) or attacker (Def. 4, and Def. 5) symbol is used twice in the attack formula, it is assumed that they refer to the same address. For example, address Vx of the first stage of attack A28 (see Table I) is the same address used by the second stage.*

Axiom 2. *It is unclear which addresses are accessed before Stage 1, hence the first accesses of both the victim and attacker may result in cache hits or misses.*

Axiom 3. *When an attacker and a victim share the same cache address space, it is assumed that a third party interference has equivalent impact on the victim's and attacker's behavior.*

Axiom 4. *The attacker and the victim do not share any memory space in the main memory.*

Axiom 5. *The attacker is able to observe the timing of the cache operations of the victim.*

Axiom 6. *The attacker knows which cryptography libraries are used by the victim. Note that a secure design entails the disclosure of the implementation [30].*

Axiom 7. *The attacker is able to force the victim to execute a specific function. For instance, the attacker can request the victim to decrypt a certain message.*

To understand how each $formula$ addresses the behavior of an attack, we will use as example the Evict+Time attack (attack ID 7). In Evict+Time attacks, the attacker reads a specific address from the cache during the victim's operation and observes if the execution time of the victim application increased. In case this happens, it means that the attacker interfered with an address used by the victim [3]. In the $formula$ of attack id 7, the attack is triggered by the victim when he accesses a cache address that is unknown to the attacker (Vx). In stage 2, the attacker accesses a certain address in the cache ($A1$), aiming to cause interference with the victim's execution by using a specific address that maps to the same cache line as Vx. In Stage 3, the victim is still using the same address used during Stage 1 (Vx). As the attacker is able to observe the time required by the victim during this stage (Axiom 5), he may identify whether the cache line address of Vx equals the one of address $A1$. Note that well-known cache attacks are summarized at the bottom of Table I.

B. Analyze Attack Models

This second step analyzes each model of the database to identify which input combinations (cache hit/miss conditions) are possible and additionally are potential attack scenarios. Each model consists of three stages in which each stage can result in a cache hit or miss. Therefore, there are theoretically eight scenarios for each attack. As a result we classify each scenario into one of the following three groups:

- Group $I \rightarrow$ *Invalid*; the combination leads to an invalid scenario. For example, it is not possible to get a cache miss during stage 2 for the attack with ID 23, as the same address is used during stage 1.
- Group $P_{\overline{Attack}} \rightarrow$ *Valid* $\cap \neg$ *Attack*; the combination leads to a valid but non-exploitable scenario.
- Group $P_{Attack} \rightarrow$ *Valid* $\cap$ *Attack*; the combination leads to a valid and exploitable scenario.

To illustrate the above, we use the Evict+Time attack (attack ID 7 in Table I) again as an example. Table II presents all the eight scenarios (cache hit/miss combinations) for this attack. The table shows in which of three groups each scenario belongs to. The last column summarizes the output of this step in the verification framework. Next we look at an example for each of the different groups. The second input combination, i.e., row 3 in Table II, is *Invalid* (Group I) because of the following reason. During Stage 2, the attacker requests data using $A1$ which result in a "hit". This means that the address used in this stage is different than Vx in Stage 1. During Stage 3, the victim requests data using Vx resulting in "miss". However, as the address of Vx was previously accessed in Stage 1 and not affected by Stage 2, a hit is expected. Hence, this combination is invalid.

TABLE II: Analysis of the attack ID 7: $Vx \rightarrow A1 \rightarrow Vx$

Vx	$A1$	Vx	I	P_{Attack}	$P_{\overline{Attack}}$	Analysis Output
hit	hit	hit	no	no	yes	$P_{\overline{Attack}}$
hit	hit	miss	yes	no	no	I
hit	miss	hit	no	no	yes	$P_{\overline{Attack}}$
hit	miss	miss	no	yes	no	P_{Attack}
miss	hit	hit	no	no	yes	$P_{\overline{Attack}}$
miss	hit	miss	yes	no	no	I
miss	miss	hit	no	no	yes	$P_{\overline{Attack}}$
miss	miss	miss	no	yes	no	P_{Attack}

```
def check_V (scenario,stage_1,stage_2,stage_3) :
  valid = True
  if (stage_1==stage_2):
        if (scenario['stage2']==MISS):
            valid = False
    elif (stage_1==stage_3):
        if (scenario['stage2']==HIT):
            if (scenario['stage3']==MISS):
                valid = False
  return valid
```

Listing 1: Python code of the valid analysis function.

The scenario of the fourth row of Table II belongs to the *Valid* $\cap \neg$ *Attack* (Group $P_{\overline{Attack}}$) because of the following reason. During Stage 2, the attacker requests data using $A1$ which results in a "miss". This means the attacker may have accessed the same address in the cache of Vx of Stage 1. Then, during Stage 3, the victim requests again the same address of Vx which now results in a "hit". This means the data requested during stage 1 is still valid in the cache. Therefore, the attacker did not interfere with the victim, thus no information is leaked.

The input combination on row 5 of Table II belongs to the *Valid* $\cap$ *Attack* (Group P_{Attack}) because of the following reasons. During Stage 2, the attacker requests data using $A1$ which results in a "miss"; hence, it fetches data from a higher memory level into cache. There is a possibility to overwrite on the address of Vx. Then, during Stage 3, the victim requests data using Vx which results in a "miss". This means that the attacker actually replaced the data on the address of Vx during Stage 2. This scenario could lead to an attack, as the attacker is able to understand which address was accessed by the victim, simply by analyzing the time it takes to complete the operation of Stage 3.

We have automated the above analysis using an algorithm for all the 28 attacks. Listing 1 presents the Python code that evaluates the invalid conditions for each scenario, while Listing 2 evaluates the conditions that lead to an access or timing attack. Note that based on our assumptions in the threat model, the interpretation of the cache attacks modeled by Deng et al. can be different. Our assumptions remove any ambiguity whether a scenario can be an attack or not.

C. Create Verification Template

The third step consists of creating a verification template, as shown in Figure 3. The first automaton (part 1) models the attack phases, as defined by the attack models in Subsection III-A. The second automaton (parts 2 and 3), models the functionality of the cache design (part 2) and the classification groups (part 3), defined in Subsection III-B. As a result, any model checker can use this template to verify whether there is a valid attack scenario or not.

```
def check_A (scenario,stage_1,stage_2,stage_3):
 attack = False
 # Access Attacks
 if (stage_3==A1)or(stage_3==Ar)or(stage_3==V1):
  if (stage_2==Vx) or (stage_2==Vr):
   if (scenario['stage3']==MISS):
     attack = True
   elif (stage_1==A1)or(stage_1==Ar)or(stage_1==V1):
    if (scenario['stage2']==MISS):
      attack = True
  elif (stage_1==Vx)or(stage_1==Vr):
   if (scenario['stage3']==MISS):
     attack = True
 # Timing Attacks
 if (stage_3==Vx)or(stage_3==Vr) :
   if (stage_1==V1)or(stage_1==A1)or(stage_1==Ar) :
     if (scenario['stage3']==MISS) :
       attack = True
 if (stage_3==Vx)or(stage_3==Vr) :
   if (stage_2==V1)or(stage_2==A1)or(stage_2==Ar) :
     if (scenario['stage3']==MISS) :
       attack = True
 return attack
```

Listing 2: Python code of the attack analysis function.

As an illustrative example, we will describe the creation of the verification template of the secure *PL cache* presented in [22]. This cache uses dynamic partitioning where each partition operates on a cache block (a group of cache lines). To control the access to each block, a lock bit is used for each cache line that verifies the process ID. In the automaton model, the hit and miss states are shared between the victim and attacker due to the dynamically partitioned cache. An internal variable "Lock" is used to represent the lock bit, which is used to check the access rights. Part 2 of Figure 3 represents this cache and the classification groups (part 3) are added in the same automaton for convenience reason. Thereafter, we integrate this automaton with the attack phases (part 1) to create the verification template, as shown in Figure 3.

To understand how these automata run, we use the first scenario that may lead to an attack in Table II (i.e., Vx=hit, $A1$=miss, Vx=miss) as example. This attack can only happen for the cache under consideration when all 3 stages are validated. First, the left automaton (part 1) starts the process by reading the inputs of the target scenario (Table II) and stores them. When the automaton traverses to stage 1, it syncs this information with the cache automaton (parts 2 and 3). Based on the first input value (Vx=hit), the cache automaton traverses to the hit state. Thereafter, the automaton traverses to the Check Lock state. In case the cache line is locked, the automaton traverses further to the bypass state and back to the initial state. In case it is not locked, the automaton traverses also back to the initial state but via the return state. Each time the automaton traverses through the bypass state it uses an internal variable to invalidate the current stage, i.e., this stage could not take place in the evaluated cache. Thereafter, the left automaton moves to the state Stage 2 and processes the second input ($A1$=miss). It again synchronizes this information with the cache automaton and the cache automaton repeats the same operations as during Stage 1. This is again repeated when the third input (Vx=miss) is processed. However, during the third stage the cache automaton traverses to part 3 of the template

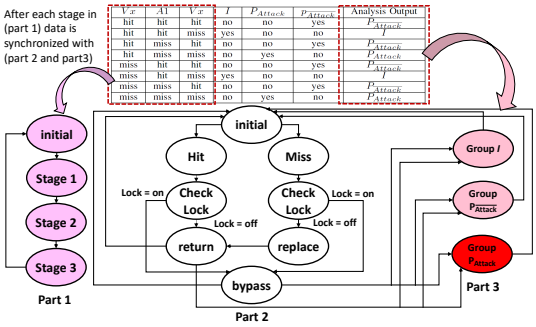

Fig. 3: Verification template of the PL Cache evaluation.

where the classification is made. When none of the stages have been invalidated, the scenario is considered valid and attack. Otherwise, it is re-classified as Group $P_{\overline{Attack}}$ (valid but not attack). Note that the scenarios classified as invalid in Table II are not evaluated and that the scenarios valid but not attack are only functionally verified.

D. Verify Vulnerabilities

The fourth and last step of the methodology verifies cache vulnerabilities, i.e., to verify whether the state Group P_{Attack} is reached or not. If the template reaches this state, it means that the model is leaking information with respect to the considered attack model and hence might not be secure. This can be formulated by reachability properties [31]. Reachability properties are used to identify whether a certain state is reachable or not from the initial state. Subsection IV-A provides more details on this property.

IV. Experimental Results

This section provides the experimental setup and results.

A. Experimental Setup

1) Cache designs: We organized the caches evaluated in this work based on their architecture model denoted as conventional (non-partitioned), static-partitioned, dynamic-partitioned and dynamic-randomization. Fig. 4 demonstrates the automata of all these cache categories. Note that for verifying the security of the cache designs only part 2 of the verification template in Fig. 3 should be replaced. For instance, for the Conventional Cache [15] part 2 should be replaced with the automaton of Fig. 4(a). Next, we briefly describe the different caches and briefly explain how we model them.

Conventional Cache [15]: The conventional non-partitioned cache is commonly used where all applications share the complete address space. Due to this, the memory space of both the common and sensitive may (partially) overlap.

SP Cache [16]: This design uses static partition and divides the cache ways into two levels: high (H) and low (L). The H way is assigned to the victim, while the L way to the attacker. The access is controlled by the process ID. To model this in the cache automaton, the attacker and victim have independent states to represent isolated accesses to each partition. A specific variable (i.e. process_id) is used to identify the victim and attackers for proper access.

SecVerilog Cache [17]: This cache design also employs static partitioning and is very similar to SP Cache. The main difference is the implementation, which uses a label instead

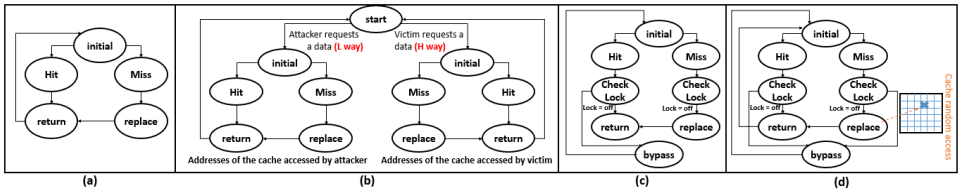

Fig. 4: Automata: (a) Conventional, (b) Static-partitioned , (c) Dynamic-partitioned, and (d) Dynamic-randomization caches.

of the process ID to control the accesses. Since we focus only on a high abstraction level, it is modelled by the same cache automaton used for SP Cache.

SecVerilog Cache v2 [18]: This cache is an extension of the previous one. Although the caches strictly do not share the ways, data might be copied from *L* to *H* and vice versa. This happens in the case both the victim and attacker access the same location in the main memory. In case one of the partitions has a cache miss, it will be copied from the other partition when available but with the same timing penalty as a cache miss. Hence, it obfuscates cache hits from another partition. Therefore, the same cache automaton model can be used here, as the behaviour at high level is the same as the previous one.

SecDCP cache [19]: This design extends the SecVerilog cache by applying dynamic partitioning. The size of *L* partition is dynamically defined by the percentage of cache misses it has. We modeled the dynamic partitions by having shared functional states that are both accessible by the attacker and victim. In the cache automaton this is modeled by transversing from the functional state to the bypass state, and subsequently returning to the initial state.

NoMO cache [20]: This dynamically partitioned cache checks access control using the thread ID. The cache automaton model of SecDCP is used to model this cache as well, as they only differ in the implementation.

DAWG cache [21]: This dynamically-partitioned cache assigns a domain ID to each process to determine which partition each process can access. The automaton of SecDCP is also used to model this cache, since they only differ in the implementation.

SHARP cache [23]: This cache uses both dynamic partitioning and randomization. It uses a lock bit called core valid bit (CVB) to control access to the victim partition. The randomization is used in the replacement policy. We modeled this special condition inside the miss and hit states by applying a random variable which may deactivate the lock bit for certain accesses.

As expected, the four categories of architectures results in four different verification templates, denoted as Conv. (conventional cache, related to Fig. 4(a)), Static (for SP, SecVerilog and SecVerilog v2 caches, related to Fig. 4(b)), Dyn (for SecDCP, NoMO, DAWG and PL caches, related to Fig. 4(c)) and Dyn-Randomization (for SHARP cache, related to Fig. 4(d)). Merging the similar templates has several benefits. First, as the approach is implementation independent, a limited number of templates are sufficient to evaluate a wide range of possible designs. Second, our evaluation provides exact details of which input combination causes the leakage. Third, the approach has a low time overhead and hence can be used to design exploration.The following subsection presents the verification results.

2) Model checker: The purpose of model checking is to verify the correctness of a system by evaluating a set of properties while traversing an automaton model of the system [32]. In this work, the cache automata are implemented in UPPAAL (Def. 10). Note that any other model checker such as NuSMV [33] or PAT [34] can be adopted for this purpose. UPPAAL [35] is a timed automata which is able to check a complex real-time model. In this work, we modeled the time using the stages of an attack formula (Def. 6).

Definition 10. *UPPAAL automaton is defined by the following Tuple* $(L, E, G, Sync, T)$*, where:*

- L *is a finite set of states,*
- E *is the set of edges that connects the states,*
- G *is the set of constraints on the edges which activate the transition between the states,*
- $Sync$ *is a set of synchronization actions which can sync the data from one automaton (denoted by o!) to another (denoted by o?),*
- T *represent the time instant,*

Based on (Def. 10), the automaton in part 1 is defined as (|L|=4, |E|=4, |G|=0, |Sync|=3, |T|=4), where L, E, G are finite set of states, edges and constraints on the edge, respectively. Sync is the synchronization channels. In this specific model checker predefined elements such as o! and o? can synchronize values between the states. T refers to the time instant. In this model checker, T is equal to execution time of each stage in the attack formula (i.e. traversing from stage 1 to stage 2 in part 1). Note that C code can be implemented in each state and thus sometimes no constraints (G parameter from the tuple, such as this illustrative example) can be seen on the edges of the automaton. For the sake of brevity, we do not describe further implementation details.

To implement the reachability property, the CTL [36] syntax equal to $E \diamond$ *PL_cache.Group*P_{Attack} is defined, where *PL_cache* is the name of the automaton, Group P_{Attack} the target state, and $E \diamond$ means "if there is an existing path that eventually reaches" the state Group P_{Attack}.

B. Results

Table III reports the analysis results. The first and 5^{th} column contain the attack ID. The remaining columns present the results of the four verification templates. Three possible outputs are shown in the table, S as secure against such attack, N as not secure against such attack, and P as partially secure against such attack. P denotes cases where the outcome is sometimes secure and other times not. This occurs for example due to randomization.

TABLE III: Security analysis of different cache models.

Attack ID	Templates				Attack ID	Templates			
	Conv.	Static	Dyn	Dyn-R		Conv.	Static	Dyn	Dyn-R
1	N	S	S	S	**15**	N	S	S	S
2	N	S	N	P	**16**	N	S	N	P
3	N	S	N	P	**17**	N	S	N	P
4	N	S	S	S	**18**	N	S	N	P
5	N	S	N	P	**19**	N	S	S	S
6	N	S	S	S	**20**	N	S	S	S
7	N	S	S	S	**21**	N	S	S	S
8	N	S	S	S	**22**	N	S	S	S
9	N	N	N	P	**23**	N	S	S	S
10	N	S	N	P	**24**	N	S	N	P
11	N	N	N	P	**25**	N	N	N	P
12	N	N	N	P	**26**	N	S	N	P
13	N	S	S	S	**27**	N	N	N	P
14	N	S	S	S	**28**	N	N	N	P

(S) denotes Secure, (N) Not Secure, and (P) Partially Secure

As observed in the table, the conventional caches are vulnerable against all attack types. The statically-partitioned caches are only vulnerable against timing attacks. Even when fully isolated, if at least the address of $V1$ is known to the attacker, a timing attack is possible as the attacker can observe the time of the third operation (Axiom 5). The dynamically-partitioned caches are less secure and are vulnerable against timing attacks and some few access attacks. The exception is the SHARP Cache [23], which employs randomization for some cache misses. The randomization used in SHARP makes the same scenario sometimes secure. As a result, we labeled this as a partially secure (letter P). One could understand P as a potential vulnerability, but less practical, i.e. the vulnerability exists but it is statistically difficult to achieve.

V. Discussion

The proposed verification methodology allows designers to evaluate for a given design whether certain attacks are possible and understand in which scenarios they occur. Hence, trade-off analysis can be made between the security and cost overhead, allowing important design decisions to be made early in the design process. For example, one could start with an SP Cache and use the information in Table III to make it more secure. Although it is very secure compared to the other cache designs, it is still vulnerable to some attacks. To improve this, our method can be used to identify which scenarios cause this. As a result, the SP Cache can be tuned for these specific cases. The vulnerabilities of SP Cache are due to cache miss of victim accesses. A countermeasure against them would be to randomly invalidate some cache lines to increase the amount of cache misses, thus adding noise to the observing attacker. This randomization could be applied only in the victim's partition; hence minimizing the timing penalty. Most of the efforts in the verification template are required in part 2 (automaton design), whereas parts 1 (stages) and 3 (classification) are automated. Therefore, the proposed verification template can be used by the semiconductor industry to evaluate the security performance trade-offs in an early design stage.

VI. Conclusion

This paper presented a formal verification template to evaluate the possibility of cache attacks for a given cache model. The template can be used to identify threats during the early design stage. The proposed methodology was demonstrated on nine different cache designs by analyzing their security using an exhaustive set of 28 models of LSCA attacks. Using this flow, designers are able to perform a design space exploration considering different countermeasures and analyze their performance and security trade-offs. '

Acknowledgments

The work has been supported by Estonian Research Council grant IUT19-1 and Estonian centre of excellence EXCITE.

References

[1] B. Gulmezoglu *et al.*, "Undermining user privacy on mobile devices using ai," in *Asia CCS '19*, 2019.
[2] D. J. Bernstein, "Cache-timing attacks on aes," Tech. Rep., 2005.
[3] Osvik *et al.*, "Cache attacks and countermeasures: The case of AES."
[4] P. Kocher *et al.*, "Spectre Attacks: Exploiting Speculative Execution," in *IEEE SP*, 2019.
[5] M. Lipp *et al.*, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium*, 2018.
[6] S. Schaik *et al.*, "RIDL: Rogue in-flight data load," in *S&P*, May 2019.
[7] M. Minkin *et al.*, "Fallout: Reading kernel writes user space," 2019.
[8] G. Doychev *et al.*, "Rigorous analysis of software countermeasures against cache attacks," *SIGPLAN Not.*, vol. 52, Jun. 2017.
[9] S. Jha *et al.*, "Towards formal verification of role-based access control policies," *Transactions on Dependable and Secure Computing*, 2008.
[10] P. Subramanyan *et al.*, "Formal verification of taint-propagation security properties in a commercial soc design," in *DATE*, March 2014.
[11] S. Deng *et al.*, "Cache timing side-channel vulnerability checking with computation tree logic," in *HASP '18*, 2018.
[12] S. Deng *et al.*, "Analysis of secure caches using a three-step model for timing-based attacks," Cryptology ePrint Archive, 2019.
[13] R. Hariharan *et al.*, "From rtl liveness assertions to cost-effective hardware checkers," in *2018 Conference on Design of Circuits and Integrated Systems (DCIS)*. IEEE, 2018, pp. 1–6.
[14] A. Danese *et al.*, "Automatic generation and qualification of assertions on control signals: A time window-based approach," in *IVLSI*, 2015.
[15] J. Handy, *The cache memory book*. Morgan Kaufmann, 1998.
[16] R. B. Lee *et al.*, "Architecture for protecting critical secrets in microprocessors," in *ISCA'05*, June 2005.
[17] D. Zhang *et al.*, "Language-based control and mitigation of timing channels," *SIGPLAN Not.*, vol. 47, Jun. 2012.
[18] D. Zhang *et al.*, "A hardware design language for timing-sensitive information-flow security," *SIGPLAN Not.*, vol. 50, Mar. 2015.
[19] Y. Wang *et al.*, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *53nd ACM/EDAC/IEEE DAC*, June 2016.
[20] L. Domnitser *et al.*, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM TACO*, Jan. 2012.
[21] V. Kiriansky *et al.*, "Dawg: A defense against cache timing attacks in speculative execution processors," in *51st IEEE/ACM MICRO*, 2018.
[22] Z. Wang *et al.*, "New cache designs for thwarting software cache-based side channel attacks," *SIGARCH Comp. Arch. News*, 2007.
[23] M. Yan *et al.*, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks," in *ISCA '17*, 2017.
[24] A. Bogdanov *et al.*, "Differential cache-collision timing attacks on aes with applications to embedded cpus," in *CT-RSA 2010*.
[25] Y. Tsunoo *et al.*, *CHES 2003*, ch. Cryptanalysis of DES Implemented on Computers with Cache.
[26] Y. Yarom *et al.*, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX*, 2014.
[27] D. Gruss *et al.*, "Flush+flush: A fast and stealthy cache attack," in *DIMVA*, 2016.
[28] O. Acıiçmez *et al.*, "Trace-Driven Cache Attacks on AES," in *Information and Communications Security*, 2006.
[29] C. Rebeiro *et al.*, "An Enhanced Differential Cache Attack on CLEFIA for Large Cache Lines," in *INDOCRYPT*, 2011.
[30] F. A. P. Petitcolas, *Kerckhoffs' Principle*. Springer US, 2011.
[31] B. Bérard *et al.*, *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.
[32] A. Sanghavi, "What is formal verification?" *EE Times_Asia*, 2010.
[33] A. Cimatti *et al.*, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, 2000.
[34] J. Sun *et al.*, "Fair model checking with abstraction." Springer, 2009.
[35] G. Behrmann *et al.*, "A tutorial on uppaal." Springer, 2004.
[36] E. M. Clarke *et al.*, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*. Springer, 1981, pp. 52–71.