LiD-CAT
A Lightweight Detector for Cache ATtacks

Reinbrecht, Cezar; Hamdioui, Said; Taouil, Mottaqiallah; Niazmand, Behrad; Ghasempouri, Tara; Raik, Jaan; Sepulveda, Johanna

DOI
10.1109/ETS48528.2020.9131603

Publication date
2020

Document Version
Accepted author manuscript

Published in
2020 IEEE European Test Symposium (ETS)

LiD-CAT: A Lightweight Detector for Cache ATtacks

Cezar Reinbrecht, Said Hamdioui, Mottaqiallah Taouil
Delft University of Technology
Faculty of EE, Mathematics and CS
m.taouil@tudelft.nl

Behrad Niazmand, Tara Ghasempouri, Jaan Raik
Tallinn University of Technology
Department of Computer Systems
jaan.raik@taltech.ee

Johanna Sepúlveda
Airbus Defense and Space
Munich, Germany
johanna.sepulveda@airbus.com

***Abstract*—Cache attacks are one of the most wide-spread and dangerous threats to embedded computing systems' security. A promising approach to detect such attacks at runtime is to monitor the System-on-Chip (SoC) behavior. However, designing a secure SoC capable of detecting such attacks is very challenging: the monitors should be lightweight in order to avoid excessive power/energy and area costs and the attack behavior should be clearly known upfront. In this work, we present LiD-CAT, a lightweight and flexible hardware detector that is aware of leakage patterns that can be used by attackers to perform cache based attacks. LiD-CAT is a cache wrapper that implements a set of leakage properties derived from cache attacks and cache models using templates. These templates identify suspicious behavior that may lead to cache attacks. LiD-CAT is evaluated using two different cache architectures, one with a secure cache and one without. On each of them, SPEC2000 benchmarks are run together with malicious applications that execute cache attacks (i.e., Evict+Time, Prime+Probe, Flush+Reload and Flush+Flush). Results show that our lightweight detector successfully detects 99.99% of the attacks with less than 1% false-positives, has no timing penalties, and increases the area of a SoC with only 1.6%.**

I. Introduction

Caches are small high-speed memory resources designed to speed up the execution of applications, including the execution of cryptographic operations [1]. Their shared nature turns cache hierarchies into a common target of microarchitectural attacks on Systems-on-Chips (SoCs), as the cache content is affected by the mutual interference of processes from different security domains. This is a major threat on SoCs where malicious and sensitive processes are executed together [2]. In this scenario, an attacker can take advantage of the mutual interference to affect the victim's process execution. Information leakage through the exploitation of cache side-channels has serious consequences on the SoC security and therefore early detection of cache attacks is critical for achieving a secure SoC. However, this task is very challenging. As the leakage is obtained solely from side-channels, the system's behavior from a security point of view is not violated. In addition, it is not fully clear which information is required to identify potential leakage and detect cache attacks. Usually these attacks are identified only after the attacker has already successfully extracted secret information or took over the control.

Although cache monitoring has been widely studied in the context of high performance SoC computation, its use for detecting cache attacks has been almost completely neglected. A limited number of publications have performed offline and online cache monitoring [3–10] for security purposes. Offline monitoring is based on the simulation of possible attack scenarios [3–6]. This technique has a very low overhead, as no additional hardware is required. However, the detection of an attack strongly depends on the coverage of the possible attack scenarios. On the other hand, online monitoring is usually based on software performance counters as in [7–10]. Despite having a higher efficiency in attack detection, the performance and cost overheads may become a limiting factor for a broad spectrum of devices.

In order to overcome such drawbacks, in this work we propose an efficient, flexible and lightweight hardware cache attack monitor for online cache attack detection. To the best of the authors' knowledge, this is the first online hardware monitor for detecting cache attacks. The lightweight property of our monitors was achieved by developing templates, which are improved, simplified and smart models which characterize the information leakage from cache side channels. Such templates exploit common cache attacks' characteristics to develop extended fingerprints capable of covering and detecting families of attacks. Moreover, these templates can easily be further customized for different cache architectures. The evaluation results demonstrate the feasibility of our approach by presenting detection results of different cache attacks, namely Evict+Time [2], Prime+Probe [2], Flush+Reload [11] and Flush+Flush [12]. In summary, the contributions of the paper are:

- Generalization of cache attack models through templates and their formal validation through assertions;
- Proposal of a new lightweight hardware monitor for detecting cache attacks;
- Evaluation of the security efficiency and performance/cost overheads of the detection mechanism for different attack scenarios.

The remainder of the paper is organized as follows: Section II presents the background concepts and the related work on cache attacks and their formal modeling. Section III describes the proposed method. Section IV presents the lightweight hardware monitor for detecting cache attacks. The experimental results are shown in Section V. Finally, Section VI discusses the limitations and concludes the paper.

II. Background

This section presents an overview of the state-of-the-art on cache attacks and cache attack models.

A. Cache Attacks

Based on the side-channel information available to the adversary, cache attacks are commonly categorized into three types: i) trace-driven attacks [13] that focus on sequences of cache operations such as hits and misses; ii) access-driven attacks [2] which exploit cache access patterns; and iii) time-driven attacks [14] which exploit the execution time. In trace-driven attacks, an adversary generates traces of the cache activity when running a victim‘s application. The traces can be obtained by measuring physical properties like power [15] or by retrieving information from system monitors [13]. An attacker can evaluate the patterns and infer sensitive information [15].

In the second attack type, access-driven, a spy process (created by the attacker) monitors whether a specific address is used by the victim's application or not. To correctly guess victim's accesses, the attacker evaluates the time required to read the target address. If the victim accesses the target address, (parts of) the spy's data is removed from the cache, producing a corresponding number of cache misses. Thus, it takes the spy longer to retrieve its data [1]. If the victim accesses cache addresses that do not collide with the target address, the spy's access to its own data is faster as all accesses result in cache hits. Using the information about the sensitive addresses of the victim and its mapping on the cache (i.e., the location of the cryptography libraries in the cache), an adversarial is able to extract secret information. Some of the well-known access-driven cache attacks are Prime+Probe [16], Flush+Reload [11] and Flush+Flush [12]. All these attacks are based on three steps: i) the attacker prepares the cache by evicting (or priming) cache sets; ii) the attacker awaits the victim's execution; and iii) the attacker re-accesses the cache and guesses which addresses were used by the victim. Prime+Probe performs the attack by reading cache addresses, Flush+Reload and Flush+Flush achieve the same objective by employing flush operations like CLFLUSH instruction from the X86 architecture.

Finally, the third attack type relies on timing-driven techniques, in which the source of leakage is the total execution time of the victim's process. Examples of time-driven attacks are the Bernstein's attack [14], Evict+Time [16] and cache-collision timing attack [17]. Bernstein's attack aims to correlate the different execution times with a guessed key. In Evict+Time, the attacker creates interference in the cache during the victim‘s operation. If the execution time changes, the attacker knows which address caused the different behavior. In collision attacks, the attacker manipulates the inputs of the victim to provoke a higher probability of cache hits. Then, the attacker reduces the key search space by selecting only the fastest encryption times.

B. Cache Attack Models

Security verification through attack simulation has been used previously to check the SoC vulnerabilities [18]. It creates a model of the system's attack surface and runs simulated attacks, which shows the SoC behavior under attack. In addition, it can be used to measure the effectiveness of protection mechanisms. However, it comes with several challenges. First, this technique tends to be very expensive due to the high hardware-software co-simulation complexity. Second, it only offers a snapshot of the SoC defenses at a particular point in time. When a SoC is updated or when new attacks are discovered, there is no way to predict how the changes will actually affect the SoC security. Third, it demands a strong knowledge of the SoC architecture, attack process and protection mechanisms. To circumvent these drawbacks, recent works have used formal techniques to model different types of cache attacks [3, 4]. These models can be used to formally verify the occurrence of a cache attack in a target SoC. The authors of [3] have introduced a three-step model to describe 28 leakage patterns (of which 20 are exploited with demonstrated attacks and 8 are not exploited yet). It includes a sequence of actions, the initiator of the actions (victim or attacker), and the information known by the attacker. These leakage patterns were further refined in [4]. By including the timing perception of each action (slow and fast cache responses) the authors were able to circumvent the adaptive time behaviour of some secure caches. Despite the wide modeling capabilities in the mentioned previous works, a straightforward implementation of such properties for online monitoring is prohibitive. In addition, these models are incomplete since they do not consider the access behavior (hit or miss). In this paper, we further refine and explore the cache leakage patterns by creating templates that understand how the attack models take advantage of target cache design. Hence, the templates can be used to perform online monitoring of the system.

III. Proposed Methodology

In this section we propose a methodology to develop lightweight online monitors based on templates of cache leakage. It first describes the inputs that are needed for the cache templates. Second, it describes how leakage properties are derived. Finally, it discusses how the templates are created.

A. Required Inputs for Cache Template Development

To develop the templates, the first step is to understand the impact of the following four inputs: i) threat model, which represents the possible vulnerabilities of the system; ii) attack model, which defines how the vulnerabilities can be exploited to conduct a cache attack; iii) target memory organization, which identifies the memory ranges where sensitive information is allocated. It allows the identification of the actors of the system (i.e., victim or attacker); and iv) cache design, which allows further customization of the leakage properties by taking the design of the cache into consideration (e.g., I/O ports, signals, addresses). Each is described in more detail below.

Threat Model: The target environment of this work is a typical Von Neumann architecture, composed by a processor, peripherals and memory hierarchy. In such a platform, an operating system manages the resources, allowing multiple processes to execute at the same time different applications. As a result, some processes share the cache memory, thus creating mutual interference. In this work we assume there are two processes named *attacker* and *victim* and their executions can lead to cache interference. We assume the following assumptions regarding the threat model:

- The main memory does not have shared space for both attacker and victim.
- The attacker has access to system's timers, being able to measure the timing of victim's execution.
- The attacker knows the public libraries the victim uses.
- The attacker can call victim's functions. For instance, the attacker can request the victim to encrypt a certain message.

Attack Model: The attack model used in this work is based on the model proposed in [3]. It contains two main actors (attacker and victim) that perform actions during three stages. Five possible actions are defined in this model:

TABLE I: Leakage Database: 28 leakage patterns 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 [2]; **b**) Cache Collision [17]; **c**) Berstein [14]; **d**) Flush+Flush [12]; **e**) Flush+Reload [11]; **f**) Prime+Probe [2]

- ***V1***: The victim accesses a cache line of which the address is known to the attacker.
- ***Vx***: The victim accesses a cache line of which the address *is not* known to the attacker.
- ***Vr***: The victim flushes a cache line of which the address *is not* known to the attacker.
- ***A1***: The attacker *accesses* a cache line.
- ***Ar***: The attacker *flushes* a cache line.

Table I summarizes the patterns that leak information. There are 28 of such patterns; 20 of them have been exploited successfully with attacks and 8 are until now not exploited. To exploit such leakages, attack models have to be created that create leakage patterns. As an example, leakage pattern *ID=7* is typically exploited by *Evict+Time* attack. An attack model is defined by a sequence of three actions that describe different cache accesses: i) attack preparation; ii) core of the attack; and iii) observation/control. For the attack *ID=7*, the attack preparation (first action) requires that the victim accesses the cache. However, the attacker *does not* need to know the victim's accessed addresses (*Vx*). Then, the core of the attack (second action) requires that the attacker reads a specific memory address (*A1*). The observation/control (third action) requires a second victim's cache access where the attacker *does not* need to know the victim's accessed addresses (*Vx*). As the attacker is able to measure the victim's execution time during the first and second accesses, an increase of the execution time will reveal that *A1* is part of the victim's accessed addresses. By repeating this process multiple times, the attacker may obtain enough information to retrieve the secret key value of table-based implementation of a cryptographic algorithm [2].

The attacker does not need any information of the victim's accesses for the attacks that are based on the leakage patterns with ID 2, 17 and 18 in Table I. That is, the actions are only constituted by *Vx* and *Vr* events. Hence, the attacker does not need to retrieve any information and in order to execute such attacks, additional trace information regarding the victim's behavior is required. Since our attack templates are based on the cache activity only, these attack types are out of the scope of this paper.

Memory Organization: Fig. 1 shows the memory regions set for the attacker's and victim's processes. As shown in this figure, the attacker and victim processes access a separate area of the main memory while the fetched address by these processes may map into the same block in the cache memory. In order to identify the process ID of an access to the cache (i.e., either the attacker or victim), information from the Operating System (OS) can be used. The OS is aware of the process *id* of each memory access and provides this to the processor. This information is also accessible by the cache.

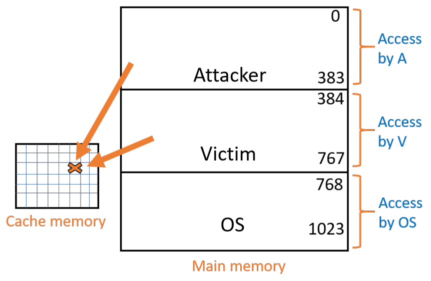

Fig. 1: Address map of the main memory.

Target Cache Designs: Two target cache designs are used in this paper. The first one is a baseline cache, which is an open-source parameterizable hardware core [19]. The configuration parameters of the cache include: i) associativity, either fully associative, direct-mapped or set-associative; ii) number of ways per cache set; iii) number of cache lines; and iv) other parameters related to memory and processor address size [20]. The second architecture is a secure cache, a protected version of [19] which was proposed in [20]. This architecture was designed to prevent access-driven side channel attacks.

B. Derivation of Leakage Properties

Formal properties are used in this work to map the attack models to a specific target cache design. To derive leakage properties, it is important to understand how the target cache design behaves during miss and hit states. The logic that generates cache hits and misses is formally described as a cache event. These events can be either cache miss or cache hit.

The next step is to identify who the trigger of an event is, i.e., the attacker or the victim. This can be realized by checking to whom the address of the event belongs. Together, the events for each of the three stages (see Table I) with their actors are used to define the leakage properties. A leakage property is an assertion that describes a leakage pattern (i.e., a combination of three events with their aggressors that matches one of the IDs in Table I) in terms of internal cache signals.

Listing 1 shows an illustrative example for the leakage property of leakage pattern *ID=7* (see also Table I). In the first step, the property checks for the event *miss or hit* (i.e., don't care) for a victim access. In the second step it looks for a *Miss* event caused by the attacker. In the third step, it looks for a *Miss* event from the victim. In the case the events with their associated actors match in all the three stages, a potential attacker is able to collect information from this leakage and hence potentially might retrieve secret information. Note that this property must be verified independently for each line of the cache during the complete execution.

C. Creation of Cache Attack Templates

The last part of the methodology is the creation of the attack templates. To achieve this objective, two tasks are required:

Listing 1: Security property for attack type ID 7.

```
property Leakage_property_ID_7;
    @(posedge clk)
    ((Miss_or_Hit && (cpu_addr inside{VICTIM_addr_range}),address_Vx = cpu_addr)
    ##[CACHE_HIT_DELAY : CACHE_MISS_DELAY]
    (Miss && (cpu_addr inside {ATTACKER_addr_range}), address_A1 = cpu_addr)  && (address_Vx == address_A1)
    ##CACHE_MISS_DELAY
    (Miss && (cpu_addr inside {VICTIM_addr_range}) && (addr_A1$_{set}$ == cpu_addr$_{set}$)))
endproperty
```

i) verification of the efficiency of the leakage properties; and ii) creation of a single Finite State Machine (FSM) that merges all leakage properties.

Verification of the Leakage Properties: To determine the effectiveness of the proposed leakage properties, we simulate all the leakage patterns identified in Table I for the two target cache designs, i.e. baseline and secure. A testbench is created that simulated all possible accesses that trigger a leakage property. For instance, for attack *ID=7*, the testbench creates a scenario where the accesses are performed by the victim, followed by the attacker and then the victim again. However, to successfully trigger the property, these consecutive accesses must result in don't care/miss/miss. Considering a specific cache line, there are multiple addresses in the main memory that could lead to such a behavior. The testbench simulates a sequence of addresses that mimic this behaviour per cache line. Finally, we evaluate how often each leakage property has been triggered when the specific scenario was applied. Depending on the property evaluated, different amount of accesses were tested, varying from 640 to 11136 combinations. Table II shows for each leakage property how often (in percentage) it has been triggered.

The baseline and secure caches are configured with a direct-mapped cache containing 64 cache lines with a 128 bit (16-byte) cache line size. For the baseline processor all leakage patterns were observed, indicating that they could be exploited by attacks. On the other hand, in the secure cache some leakage patterns were not triggered at all or with a lower activity. For instance, in case 7 (security property 7) only 50% of the leakage patterns was observed. This is due to the protection technique based on address randomization, which obfuscates the relation between the address in the cache line and main memory from the attacker. For two cases, i.e. case 1 and case 21 (Prime+Probe attack), the leakage properties did not occur, which means that the countermeasure functioned properly and mitigated such leakage patterns. For the other cases, i.e. the leakage patterns that are not related to access-driven attacks, the secure cache resulted in the same amount of triggers as the baseline cache. This implies that there is no protection against these leakage patterns.

Creation of the Finite State Machine: The leakage properties are combined to create a single finite state machine. To represent the states, we employ the actions from the leakage patterns of Table I, namely V1, Vx, Vr, A1 and Ar. All these five actions can be used by an attacker as the first step of an attack. This means that the first stage has five states. Thereafter, each state may lead again to five states in the second stage. However, not all the combinations are possible. The leakage properties determine which sequence of actions can be exploited. The resulting cache template FSM is presented in Fig. 2. Note that the state machine contains at each stage both the actors and the cache event.

TABLE II: Evaluation of triggered leakage properties.

	Cache Configuration			**Cache Configuration**	
Leakage prop.	64/1/16 (baseline)	64/1/16 (secure)	Leakage prop.	64/1/16 (baseline)	64/1/16 (secure)
Case_1	100%	0%	Case_15	100%	100%
Case_2	N/A	N/A	Case_16	100%	100%
Case_3	100%	100%	Case_17	N/A	N/A
Case_4	100%	82.7%	Case_18	N/A	N/A
Case_5	100%	100%	Case_19	100%	91.6%
Case_6	100%	94.2%	Case_20	100%	100%
Case_7	100%	50%	Case_21	100%	0%
Case_8	100%	100%	Case_22	100%	100%
Case_9	100%	100%	Case_23	100%	100%
Case_10	100%	100%	Case_24	100%	100%
Case_11	100%	100%	Case_25	100%	100%
Case_12	100%	100%	Case_26	100%	100%
Case_13	100%	50%	Case_27	100%	100%
Case_14	100%	100%	Case_28	100%	100%

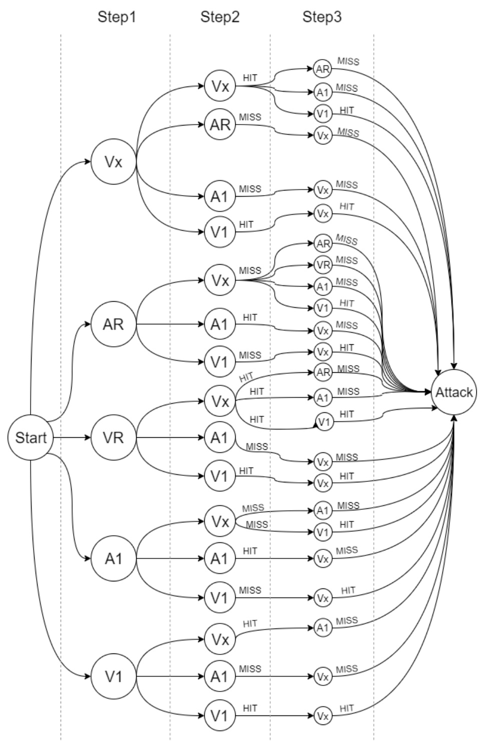

Fig. 2: Final leakage template.

The FSM of Figure 2 can be optimized by grouping together common states to create a lightweight implementation. For example, *V1* and *Vx* can be grouped as the hardware is not aware if the accessed address is known to the attacker or not. Our optimization reduces the number of states to only eight states, as shown in Fig. 3. Hence, each cache line requires three state bits.

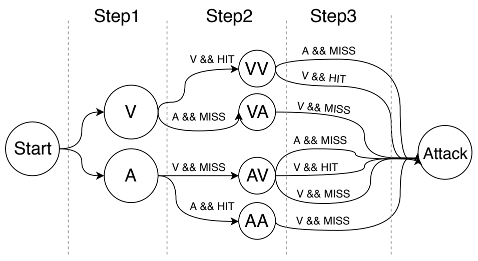

Fig. 3: Optimized FSM implemented in LiD-CAT.

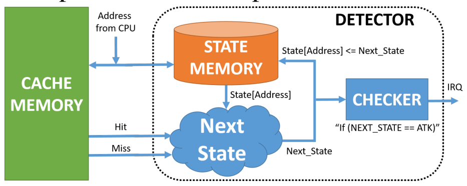

Fig. 4: LiD-CAT architecture.

IV. The LiD-CAT Architecture

In this section we introduce the LiD-CAT architecture and its integration within the cache design.

A. Design

Fig. 4 shows the architecture of LiD-CAT. It consists of four components: i) state memory, a dedicated memory (e.g., a register file) to store the FSM states of each cache line; ii) next state logic, which defines the FSM transitions of Fig. 3; iii) hit and miss monitors, to track events inside the cache; and iv) checker, that verifies whether the access patterns match the FSM states, hence it checks whether leakage takes place or not.

LiD-CAT operation is composed of several steps. First, the detector awaits for a memory request from the processor. Thereafter, the detector fetches from the state register file the state related to the accessed cache line. LiD-CAT uses the cache status information (i.e., hit or miss given by the hit and miss monitors), the address, and the current state to determine the next state. If the current request matches a step from the leakage pattern, the FSM will traverse to next state. When all states match a certain leakage pattern, the checker triggers an interruption flag (i.e., IRQ). Otherwise, the FSM jumps to one of the states at Step 1 (see Fig. 3) based on the cache access' actor (victim or attacker). Finally, the output of the next state logic is updated in the state memory.

The LiD-CAT detector identifies the behavior of all leakage patterns defined by the final leakage template (see Fig. 2). They are comprised of a sequence of events that may affect any cache line. To build a LiD-CAT detector, two tasks are required: i) to analyze the correct sequence of accesses; and ii) to evaluate each cache line independently. To accomplish these tasks, the LiD-CAT architecture is based on multiple FSMs (see Figure 2), each evaluating a single cache line.

B. Integration

To integrate LiD-CAT into a cache, cache hit and miss monitors are required. These monitors can be integrated in an invasive or non-invasive (without modifying the cache architecture) way. The target cache architectures described in [19] and [20] allow access of the internal signals of the cache, as shown in Fig. 4. The monitors of LiD-CAT compare through an AND gate two internal signals of the target cache design: i) request from the CPU; and ii) hit flag from the cache. The miss signal is generated by inverting the hit signal. Note that the monitoring hardware overhead is not significant. However, the designer must ensure the meeting of the timing constraints, since depending on the way the monitors are integrated, the critical path may be affected.

The current LiD-CAT architecture uses registers to store the states. However, the cache memory itself can be used. In that particular case each cache line would be extended with three extra state bits.

V. Experimental Results

In this section, we present the performance, cost and security results of LiD-CAT.

A. Setup

LiD-CAT is modelled in Verilog and integrated into two cache designs: i) a baseline cache [19]; and ii) a secure cache, which is the protected version of [19] against access-based cache attack [20]. The goal is to evaluate its detection accuracy (i.e., both false and true positives) and hardware overhead. To realize this, our experiments contain both non-malicious and malicious applications. The non-malicious applications are selected from SPEC2000 CPU Benchmark; they are Swim, Gzip, Gcc and Mcf [21]. As malicious applications popular attacks have been used; they are the Evict+Time (E+T) [2], Prime+Probe (P+P) [2], Flush+Reload (F+R) [11] and Flush+Flush (F+F) [12]. All attacks are performed on the transformation table (T-table) implementation of the Advanced Encryption System (AES) with a 128-bit key. In all experiments 10000 memory accesses per application are evaluated.

To evaluate the hardware overhead of LiD-CAT, it is integrated in an SoC and synthesized on a Cyclone IV GX FPGA from Intel. The target SoC comprises a NIOS II processor [22], a cache design, flash memory controller, DRAM memory controller, UART serial interface, JTAG interface, Direct-Memory Access (DMA), timer, Ethernet with MAC, and a bus interconnection. The SoC was synthesized for both the baseline cache and secure cache.

B. Detectors Accuracy

The top part of Table III shows the detection accuracy of LiD-CAT. The false positives results correspond to the ratio of triggers of leakage properties that LiD-CAT detects for non-malicious applications. The table shows that the highest amount of false positives occurs for the baseline cache which goes up to 14% for gzip. Gzip reads the same memory addresses periodically and hence LiD-CAT identifies such behavior as a possible timing leakage (see Table I *ID=8* to *ID=12*). For the secure cache, the false positive results are typically lower and the worst case reaches 11% for mcf. The main reason is due to the protection mechanism of the secure cache of [20]. It masks the accesses in the cache by using random addresses, thus changing the access behavior of Gzip. False-positives are not desired in a system as the false alarms can create several performance drawbacks due to interruption handling. Therefore, to reduce the amount of false-positives, we added a threshold trigger to LiD-CAT that creates an alert (Interupt Request in Fig. 4) when a certain amount of triggers are accumulated. The threshold should be

TABLE III: Experimental results of LiD-CAT.

False-Positive Results - SPEC2000 Benchmarks				
Cache Design	swim	gzip	gcc	mcf
Baseline	4.97%	14.08%	5.87%	10.18%
Secure	7.87%	7.79%	7.2%	11.24%
With threshold of 100 triggers				
Baseline	0.04%	0.14%	0.05%	0.10%
Secure	0.07%	0.07%	0.07%	0.11%
True-Positive Results - Cache Attacks				
Cache Design	E+T	P+P	F+R	F+F
Baseline	100%	100%	100%	100%
Secure	100%	100%	100%	100%
With threshold of 100 triggers				
Baseline	99.99%	99.99%	99.99%	99.99%
Secure	99.99%	99.99%	99.99%	99.99%
Synthesis Results - Area Overhead				
Cache Design	LC Comb	LC Reg	Memory	% of SoC
Baseline	21.5%	19.6%	0%	1.58%
Secure	10.1%	11.4%	0%	1.46%
With threshold of 100 triggers				
Baseline	22.4%	20.3%	0%	1.65%
Secure	10.5%	12.8%	0%	1.52%

high enough to minimize false positives but low enough to still detect the leakage patterns effectively. As T-Table based AES encryption requires 160 cache memory accesses [14], a threshold of 100 triggers is selected. With this threshold, the amount of false-positives decreases below 0.14%. The true positives correspond to the amount of triggered LiD-CAT detection (possible attack) for malicious applications. Without the trigger, all attempts of the attacker to create one of the leakage patterns used in E+T, P+P, F+R and F+F were detected. With the threshold enabled, the detection accuracy only reduced with 0.01%. Considering the benefits of significantly reducing the false-positives while almost not affecting the detection of true-positives, we can conclude that the use of a threshold is an effective strategy.

C. Hardware Overhead

The synthesis results of the cache designs, LiD-CAT (LC) and the SoC are presented in the bottom part of Table III. The cache designs include an SRAM memory array together with additional logic to manage hits and miss behaviors. The area of this logic is typically very small compared to the complete SoC. Hence, we compare the area overhead of LID-CAT to the complete SoC. For both the baseline and secure cache, LiD-CAT without the threshold trigger has an area overhead of 1.52%. With the threshold trigger, it increase to 1.65%, due to the need of an additional 7-bit counters that counts to 100.

VI. Discussion and Conclusion

This paper presented LiD-CAT, a lightweight online detector of cache attacks. From the paper we conclude:

Threat Response: After the identification of a leakage pattern, the system has to make a decision on how to mitigate the threat. This action may impact the performance. One option is to flush the cache entirely to erase the access history. Another option would be to block other processes until the victim finishes. A third option would be to randomize the memory mapping to confuse the attacker.

Multi-port Caches: In the case the cache has multi-port access, LID-CAT can be easily modified. It requires to have next stage logic and checkers for each port, and to have a multi-port state memory (see Fig. 4)

Threshold Limitation: Reducing the threshold trigger has limitations for sensitive applications. For example, T-Table AES requires 160 accesses for one encryption. This means that the threshold trigger must be set below 160 triggers. If the value is set higher, an attacker can observe the whole encryption, and after that, restart the system and perform the attack again with different inputs. Hence, the false-positives cannot be completely removed.

Multiple Thresholds: LiD-CAT uses only one counter to implement the threshold trigger, as only one leakage pattern can be exploited at a time. However, an attacker might still be able to create sophisticated attacks to bypass the LiD-CAT detector. In such a scenario, a different counter should be used for each leakage pattern. Due to the limited number of leakage patterns, it comes with a marginal overhead.

Design Verification: Our experiments show that the secure cache, compared to the baseline, avoided up to 60% of all leakage patterns in general and 100% of leakage patterns belonging to access-based cache attacks. Hence, our methodology can also be used statically at design time to evaluate the severeness of the information leakage of caches and verify the design from a security point of view.

Acknowledgments

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

References

[1] J. Sepulveda *et al.*, “Exploiting bus communication to improve cache attacks on systems-on-chips,” in *ISVLSI*, 2017.
[2] D. A. Osvik *et al.*, *Cache Attacks and Countermeasures: The Case of AES.* Springer Berlin Heidelberg, 2006.
[3] S. Deng *et al.*, “Cache timing side-channel vulnerability checking with computation tree logic,” in *HASP ’18*, 2018.
[4] S. Deng *et al.*, “Analysis of secure caches using a three-step model for timing-based attacks,” *HaSS*, 2019.
[5] Z. He and R. B. Lee, “How secure is your cache against side-channel attacks?” in *MICRO*, 2017.
[6] T. Ghasempouri *et al.*, “Security verification template to assess cache architecture vulnerabilities,” in *DDECS*, 2020.
[7] M. Chiappetta *et al.*, “Real time detection of cache-based side-channel attacks using hardware performance counters,” *ASC*, 2016.
[8] S. Briongos *et al.*, “Cacheshield: Protecting legacy processes against cache attacks,” *CoRR*, 2017.
[9] M. Mushtaq *et al.*, “Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters,” in *HASP*, 2018.
[10] T. Zhang *et al.*, “Cloudradar: A real-time side-channel attack detection system in clouds,” 2016.
[11] Y. Yarom and K. Falkner, “Flush+reload: A high resolution, low noise, l3 cache side-channel attack,” in *23rd USENIX*, 2014.
[12] D. Gruss *et al.*, “Flush+flush: A fast and stealthy cache attack,” in *DIMVA*, 2016.
[13] O. Acıiçmez and Çetin K. Koç, “Trace-driven cache attacks on aes,” in *ICICS*, 2006.
[14] D. J. Bernstein, “Cache-timing attacks on aes,” Tech. Rep., 2005.
[15] J.-F. Gallais *et al.*, “Improved trace-driven cache-collision attacks against embedded aes implementations,” in *WISA*, 2011.
[16] E. Tromer *et al.*, “Efficient cache attacks on aes, and countermeasures,” *Journal of Cryptology*, 2010.
[17] A. Bogdanov *et al.*, “Differential cache-collision timing attacks on aes with applications to embedded cpus,” in *CT-RSA 2010.*
[18] S. Chattopadhyay and A. Roychoudhury, “Symbolic verification of cache side-channel freedom,” *TCAD*, 2018.
[19] Chair of VLSI Design, Diagnostics and Architecture. (2016) PoC - Pile of Cores. Technische Universität Dresden. [Online]. Available: https://github.com/VLSI-EDA/PoC
[20] B. Niazmand *et al.*, “Design and verification of secure cache wrapper against access-driven side-channel attacks,” in *DSD*, 2019.
[21] S. Sair and M. Charney, “Memory behavior of the spec2000 benchmark suite,” Technical report, Tech. Rep., 2000.
[22] I. F. Altera, “Nios ii classic processor reference guide,” Available at: https://www.altera.com/, note = Accessed at 2017-10-07,, 2015.