Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

**Technical Report No. NJU-SEG-2023-IC-003**

**2023-IC-003**

# SCAGuard: Detection and Classification of Cache Side-Channel Attacks via Attack Behavior Modeling and Similarity Comparison

Limin Wang, Lei Bu, Fu Song

Technical Report 2023

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247890

# SCAGuard: Detection and Classification of Cache Side-Channel Attacks via Attack Behavior Modeling and Similarity Comparison

Limin Wang[1], Lei Bu[1(✉)], and Fu Song[2]
[1]*State Key Laboratory of Novel Software Techniques, Nanjing University, Nanjing, Jiangsu 210023, China*
[2]*School of Information Science and Technology, ShanghaiTech University, Shanghai 201210, China*
Email: wanglimin@smail.nju.edu.cn, bulei@nju.edu.cn, songfu@shanghaitech.edu.cn



***Abstract*—Cache side-channel attacks (CSCAs), capable of deducing secrets by analyzing timing differences in the shared cache behavior of modern processors, pose a serious security threat. While there are approaches for detecting CSCAs and mitigating information leaks, they either fail to detect and classify new variants or have to impractically update deployed systems (e.g., CPU). In this work, we propose a novel approach, named SCAGUARD, to detect and classify CSCAs via attack behavior modeling and similarity comparison. Specifically, we introduce the notion of cache state transition enhanced basic block sequences (CST-BBSes) to model attack behaviors which is able to capture both attack-relevant syntactic code information and semantic cache information. We propose an approach to automatically construct CST-BBS models from binary programs. To detect and classify attacks, we adapt a dynamic time warping algorithm to compare the similarity of CST-BBSes between attack and target programs. We implement our approach in a tool SCAGUARD and evaluate it using real-world attacks and diverse benign programs. The results confirm the effectiveness of our approach, compared over existing detection approaches. In particular, SCAGUARD significantly outperforms the other detection approaches on new variants.**

## I. Introduction

*Cache side-channel attacks* (CSCAs), e.g., Flush+Reload [1] and Prime+Probe [2], are able to effectively exploit the timing difference caused by access patterns (e.g., cache hit and cache miss) of shared CPU caches to infer secrets within the same physical device. To thwart CSCAs, various mitigation approaches (e.g., constant-time techniques and novel cache architectures [3]) have been proposed to break the dependence between secret and timing difference of cache access, thus eliminating cache side-channel vulnerabilities. Though promising, these approaches have to update deployed software and/or hardware systems, hence are difficult to quickly apply to existing systems. Instead of eliminating vulnerabilities, detection approaches are proposed to block CSCAs without updating deployed software and hardware systems. To detect and classify attacks, existing approaches either use machine learning (e.g., [4], [5]) or heuristic rules (e.g. [6]). The former requires a large set of training samples of running data from the attacker for training, which are difficult to collect due to the lack of high-quality cache attack samples, moreover, often fails to identify new variants that are not included in the training data. The latter relies on manually designed patterns of existing CSCAs, hence are not flexible and can be easily bypassed by new variants.

To overcome the drawbacks of existing CSCA detection approaches, in this work, we propose a novel attack-oriented detection approach, named SCAGUARD. SCAGUARD automatically builds attack behavior models from the Proof-of-Concepts (PoCs) of existing attacks. For each target program, SCAGUARD compares the similarity degree between the behavior models of the target and attack programs. If the similarity degree is high, the target program is regarded as a variant of the attack program.

This work is partially supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001), the Fundamental Research Funds for the Central Universities (No.2022300291), and the National Natural Science Foundation of China (Nos. 62232008, 62172200, 62032010 and 62072309).

TABLE I: HPC EVENTS USED IN THIS WORK

| Scope | Event |
|---|---|
| L1 Cache | L1 Data Cache Load Miss, L1 Data Cache Load Hit, L1 Data Cache Store Hit, L1 Instruction Cache Load Miss |
| LLC | LLC Load Miss, LLC Load Hit, LLC Store Miss, LLC Store Hit |
| Others | Branch Miss, Branch Load Miss, Cache Miss, Timestamp |

Specifically, the control flow graph (CFG) is a good candidate for modeling attack behaviors. However, solely modeling the attack behavior of an attack program as a CFG is neither accurate due to a large number of attack-irrelevant basic blocks nor robust due to pure syntactic code information and various ways to implement an attack. Therefore, we propose to locate attack-relevant basic blocks and eliminate attack-irrelevant basic blocks in the CFG by leveraging runtime execution information. To capture semantic cache information, we propose to enhance the reduced control flow graph with cache state transition, resulting in the attack behavior model, called *cache state transition-enhanced basic block sequence* (CST-BBS). The CST-BBS model is able to capture both attack-relevant syntactic code information and semantic cache information. To check if a target program is a variant of an existing attack, SCAGUARD automatically constructs CST-BBS models from their binary implementations and compares the similarity degree between the CST-BBS models by adapting a Dynamic Time Warping algorithm [7].

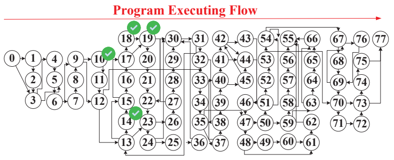
To evaluate our approach SCAGUARD, we implement it in a tool and conduct experiments using 2800 benchmarks consisting of 400 attack programs for each type of CSCAs (Flush+Reload Family, Prime+Probe Family, as well as their Spectre-like variants and obfuscated variants) and 400 diverse benign programs. The experimental results show that our approach can accurately build attack behavior models and identify attack variants. For instance, the detection precision of SCAGUARD is 96.64% which is better than the state-of-the-art approaches. More importantly, on new attack variants (Spectre-like variants, other attack family's variants, or obfuscated variants) that have not been used in attack behavior modeling, SCAGUARD is still able to achieve more than 90% detection precision, 3.25%–95.2% higher than that of the state-of-the-art approaches.

In summary, the main contributions of this work are:

- We introduce an attack behavior model called CST-BBS and propose an approach to automatically build CST-BBS models of binary programs, for capturing both attack-relevant syntactic code information and semantic cache information.
- We present a Dynamic Time Warping based algorithm for measuring the similarity of CST-BBS models which allows us to detect and classify attack variants.
- We implement our approach in a tool and conduct a thorough evaluation on a large set of programs including Flush+Reload Family, Prime+Probe Family, and their spectre-like variants. The results confirm the efficacy of our approach.

```
Require: The attacker-chosen shared memory addresses addr;
1: /* the Flush step*/
2: clflush(addr)
3: /* the Reload step*/
4: time1 = RDTSCP();
5: Read addr;
6: time2 = RDTSCP();
7: access time=time2 - time1;
```

(a) The sketch code of the Flush+Reload attack




(b) The CFG of a proof-of-concept Flush+Reload attack, where the attack-relevant BBs are marked in green color.

Fig. 1: The sketch code and the CFG of the Flush+Reload attack

## II. Background

In this section, we first introduce basic concepts and then recall typical cache side-channel attacks.

### A. Preliminaries

**Control flow graph.** A *basic block* (BB) is a straight-line sequence of instructions with no branches in except to the entry and no branches out except at the exit, where each instruction is associated with its instruction address.

***Definition 1:*** Given a program $P$, the control flow graph (CFG) $G$ of $P$ is a tuple $(V, E)$, where $V$ is a set of nodes, each of which represents a BB, and $E \subseteq V \times V$ is a set of directed edges, each of which represents the control flow from one BB to another one.

**Hardware performance counters (HPCs).** As mentioned above, solely modeling the attack behaviors via CFG is not accurate. Thus, we leverage HPCs, available in modern processors (e.g., Intel, AMD, and ARM), for monitoring and measuring CPU-related events (e.g. instruction retired, cache hit/miss, etc.) during process execution [8]. In this work, we will use HPCs listed in Table I to collect the cache hit/miss events in Level-1 Cache (L1), Last Level Cache (LLC), the branch-related information, and the timestamp.

**Cache state transition enhanced basic block sequence (CST-BBS).** To capture semantic cache information, we introduce CST-BBS.

***Definition 2:*** The *occupancy rate* in the cache is the ratio of non-empty cache lines to the total number of cache lines.

***Definition 3:*** A *cache state* $S$ at a program point in an execution is a tuple $(AO, IO)$, where $AO$ denotes the occupancy rate of the cache lines by the attack program and $IO$ denotes the occupancy rate of the cache lines excluding those occupied by the attack program. Clearly, $AO + IO \leq 1$ for any cache state $(AO, IO)$.

***Definition 4:*** A *cache state transition* (CST) of a BB $b$ is a tuple $(S, b, S')$, denoted by $S \xrightarrow{b} S'$, such that the execution of the BB $b$ under the cache state $S$ yields the cache state $S'$.

***Definition 5:*** Given a sequence $b_1, b_2, \cdots b_n$ of basic blocks (BBS), a CST-BBS of the BBS $b_1, b_2, \cdots b_n$ is a sequence of cache state transitions $S_1 \xrightarrow{b_1} S'_1$, $S_2 \xrightarrow{b_2} S'_2$, $\cdots$, $S_n \xrightarrow{b_n} S'_n$.

### B. Cache Side-Channel Attack (CSCA)

Two well-known CSCA families [1], [2] and their variants combined with the new microarchitecture attacks [9], [10] are as follows:

**Flush+Reload Family.** The Flush+Reload family mainly contains Flush+Reload and its variants Evict+Reload, Flush+Flush, all of which rely on a shared memory (usually a shared library).

The sketch code of the Flush+Reload attack [1] is shown in Fig. 1 (a) and its corresponding CFG is shown in Fig. 1 (b), the attack-relevant BBs are marked in green color. The detailed code and CFG refer to [11]. It consists of the following two key steps. (i) Flush step: the attack first flushes the chosen memory blocks (Fig.1(a) line 2, BB 10 in Fig.1(b)) via the X86 *clflush* instruction. Then if the victim accesses the same memory blocks next time, these blocks will be fetched back to the caches. (ii) Reload step: the attack re-loads the *chosen* memory blocks (Fig.1(a) line 5, BBs 15–17 in Fig.1(b)) and computes the access time of the reloading operation (Fig. 1(a) lines 4, 6 and 7, BBs 14, 18 and 19 in Fig. 1(b)), where the function *RDTSCP* obtains the current time stamp. If the access is faster than a pre-set threshold, it is presumed that a cache hit occurs, which indicates that the current accessed memory addresses have been accessed by the victim. In this way, the attack can obtain cache access patterns of the victim based on which the adversary may deduce the victim's secret.

Unlike Flush+Reload, Evict+Reload [12] evicts the corresponding cache set of the chosen shared memory addresses by loading the attack's data instead of clflush-like instructions, and Flush+Flush [13] exploits the time difference of clflush instruction execution that caused by data being cached or not, rather than the time difference in cache hits/misses, to obtain the victim's cache access pattern.

**Prime+Probe Family.** Prime+Probe [2] is the most widely-used attack in Prime+Probe Family which does not need the shared memory between the attacker and the victim. It consists of the following two key steps: (i) Prime step: the attack fills the corresponding cache set of the chosen memory addresses using its own data. (ii) Probe step: the attack re-accesses the *same* memory addresses and measures the access time. If the access is slow, it is presumed that a cache miss occurs, indicating that the cache lines have been evicted by the victim.

**Variants of CSCAs.** Classic CSCAs would become ineffective if the cache access pattern of the victim is independent of the secret without transient execution. However, by exploiting branch prediction and out-of-order execution respectively, Meltdown [9] and Spectre [10] can illegally access out-of-bound memory addresses, and when some unauthorized secret data is cached, the attacker are still able to infer them through existing CSCAs.

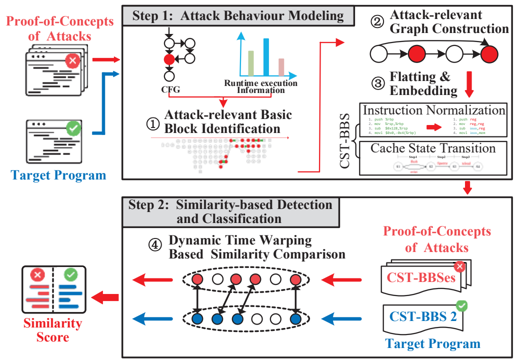



Fig. 2: The workflow of the proposed CSCAs detection approach SCAGuard

## III. Methodology

In this section, we present the details of our approach SCAGuard. Fig.2 illustrates its overview that consists of two key steps: 1) Attack behavior modeling, 2) Similarity-based detection and classification.

### A. Attack Behavior Modeling

We first present a runtime data driven method to identify attack-relevant BBs from the CFG of a given program. We then construct an attack-relevant graph from the CFG and enhance it with cache state transitions, yielding an attack behavior model.

*1) Attack-relevant BB Identification:* Consider the Flush+Reload shown in Fig. 1 (b). We can observe that many BBs are attack-irrelevant (i.e., the BBs not highlighted in the green color), and only a few of them are attack-relevant. Therefore, solely using CFG to represent an attack behavior would not be precise enough to detect and classify attacks due to a large number of attack-irrelevant BBs. To solve this problem, given a PoC, we first build its CFG by utilizing off-the-shelf tools (e.g., Angr [14] in our implementation)

**Algorithm 1** Attack-relevant graph construction

**Input:** A control flow graph $G$, a set of attack-relevant BBs $N$
**Output:** Attack-relevant graph $G_A$
1: $G$= RemoveCycle($G$);
2: Attach the HPC data to each node in $G$;
3: **for** each pair of nodes $v_i, v_j, \in N$ **do**
4: $P_{i,j}$ =all the paths between $v_i$ and $v_j$ in G without going through any other attack-relevant BBs;
5: $G'$ =AttackCorrelationEvaluation($P_{i,j}$);
6: **end for**
7: $G''$=the maximum spanning tree of $G'$ ;
8: Restore all the nodes and edges of $G''$ to $G_A$;
9: **return** $G_A$;

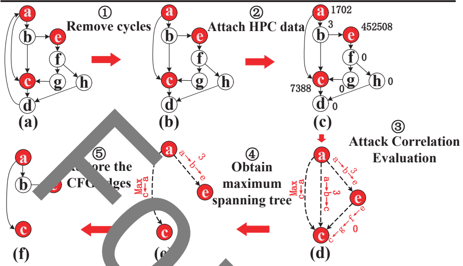



Fig. 3: The steps of the attack-relevant graph construction

and identify potential attack-relevant BBs in the CFG by leveraging runtime information in two steps.

In the first step, we collect HPC data with corresponding instruction addresses by executing the PoC using tracing tools, e.g., perf-intel-pt [15] in our implementation for Intel processors. We then map all the HPC data to the BBs in the CFG according to instruction addresses of the HPC data and BBs, based on which we compute the HPC value of each BB which is the sum of the selected 11 HPC events (excluding the timestamp) shown in Table I. A BB with non-zero HPC value is regarded as a potential attack-relevant BB, as the BB contains instructions that conducted cache-related operations.

In the second step, we eliminate further more attack-irrelevant BBs using cache access information based on the following key observation. During a cache side-channel attack, some cache sets must be accessed multiple times and there exist at least two BBs that access overlapped cache sets. Consider the Flush+Reload attack, the sets of cache sets accessed by the BBs of the Flush and Reload steps are $X = \{3, 4, 5, 8, 15\}$ and $Y = \{3, 4, 5, 8, 10, 24, 34, 50, 77, 78\}$, respectively, leading to $X \cap Y = \{3, 4, 5, 8\}$. Based on this observation, we collect the accessed memory addresses (including flushed addresses) of each potential attack-relevant BB obtained in the first step. This is done by utilizing Intel PT [16] in our implementation for Intel processors. Then, we identify the cache sets that are accessed by multiple BBs and eliminate the BBs that do not access any memory addresses corresponding to the multiply-accessed cache sets.

*2) Attack-relevant Graph Construction:* To further build the attack behavior model, we propose to construct an attack-relevant graph by connecting all the identified BBs with the most possible attack-relevant paths from the original CFG. Intuitively, we choose a path between each pair of attack-relevant BBs with the highest average HPC value as the most possible attack-relevant path. All such paths are merged together, leading to an attack-relevant graph. This attack-relevant graph contains the paths that are highly correlated with the attack behavior and also covers some attack-relevant BBs that may have been eliminated due to the lack of cache access operations, but conducted necessary operations for the attack.

Our idea is formalized in Algorithm 1. We exemplify it using an example whose CFG is shown in Fig. 3 (a), where the attack-relevant blocks are highlighted in red color.

1) First of all, in order to make the attack-relevant graph loop-free, cycles/loops in the CFG $G$ are eliminated by removing the backward edges (line 1, Algorithm 1). For example, we eliminate the cycle $a \to b \to c \to d \to a$ in Fig. 3 (a) by deleting the backward edge $d \to a$, resulting in the CFG shown in Fig. 3 (b).
2) We then attach the HPC values to all the BBs for attack correlation evaluation (line 2), leading to the CFG shown in Fig. 3 (c).
3) In lines 3-5, we build a directed, weighted graph $G'$ as follows. For each pair of attack-relevant BBs $v_i, v_j \in N$, we compute all the paths between $v_i$ and $v_j$ in the CFG $G$ that do not go through any other attack-relevant BBs, forming the set $P_{i,j}$ (line 4). For each path $p = v_i \to v_{i+1} \to \cdots \to v_{j-1} \to v_j$ in $P_{i,j}$, we evaluate its attack correlation value $V_p$ as the average HPC value of all BBs in the path $p$ excluding the endpoints $v_i$ and $v_j$, i.e.,
$$V_p = \begin{cases} \frac{1}{j-i-1}\sum_{k=i+1}^{j-1}\texttt{HPC}(v_k), & \text{if } j > i+1;\\ \texttt{MAX}, & \text{if } j = i+1.\end{cases}$$
where $\texttt{HPC}(v_k)$ is the HPC value of the BB $v_k$ and $\texttt{MAX}$ a large enough value. Specially, if $v_i$ and $v_j$ are directly connected, $V_p = \texttt{MAX}$. Then, we add an edge $v_i \to v_j$ into $G'$ which is labeled by $(p, V_p)$ (line 5). For example, in Fig. 3(c), there are two paths $a \to b \to c$ and $a \to c$ connecting $a$ and $c$ that does not goes through any other attack-relevant BBs. Thus, $a \to c$ with labels $(a \to b \to c, 3)$ and $(a \to c, \texttt{MAX})$ are added (cf. Fig. 3 (d)).
4) Since the higher the value $V_p$ of a path $p$, the higher the probability that $p$ is correlated with attack behaviors, to find the most possible attack-relevant paths, we take $V_p$ as the weight and compute the maximum spanning tree (MST) $G''$ of the weighted graph $G'$ (line 7) using the MST algorithm [17]. $G''$ connects all the attack-relevant BBs with the maximum weights. For the example in Fig. 3 (d), we obtain the MST shown in Fig. 3 (e).
5) Finally, for each edge in the MST $G''$, the labeled path $p$ is restored, namely, the edges and nodes in the path $p$ are added into a new directed graph $G_A$ that is used as the attack-relevant graph. For example, the labeled path $a \to b \to e$ of the edge $a \to e$ in Fig. 3 (e) is restored, and two edges $a \to b$ and $b \to e$ are added into the attack-relevant graph as shown in Fig. 3 (f).

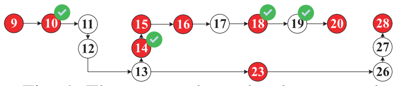



Fig. 4: The extracted attack-relevant graph

By Algorithm 1, we can build an attack-relevant graph which includes all the potential attack-relevant BBs and their control flows. For the Flush+Reload example, Fig. 4 shows its attack-relevant graph obtained from the CFG in Fig. 1 (b), where the potential attack-relevant BBs are highlighted in red color, covering all the manually identified attack-relevant BBs highlighted by green checkmarks.

*3) Attack Behavior Model Construction:* Due to the diversity of attack variants, similar attack behavior of the attack programs may have different pure syntactic code information, which makes them look dissimilar with each other. Therefore, it is important to embed semantic cache information in attack behavior models to detect attack variants. To do so, we propose to enhance each BB in the attack-relevant graph with a CST, thus capturing semantic cache information.

To measure the CST, w.l.o.g., we set a specific scenario for the simulation of each BB. In this scenario, initially, the cache is full of data and the attack is not mounted, that is $IO$= 1, and $AO$= 0. Then, we simulate each attack-relevant BB by feeding the accessed memory addresses of the instructions in the BB (collected in Section III-A1) into a cache simulator (e.g., [18]), and observe the decreasing of $IO$ and the increasing of $AO$ to obtain the corresponding CST for the BB, which captures the semantic cache information. Finally, we flatten the attack-relevant graph into a BBS according to the execution timestamp of each BB and embed the collected CSTs into BBS, resulting in a CST-BBS, which models the attack behavior of PoC.

TABLE II: THE ATTACK DATASET

| Abbr. | Type | Samples[1] | #C[2] | #M[3] |
|---|---|---|---|---|
| FR-F | Flush+Reload (FR) Family | FR-IAIK, FR-Mastik, FR-Nepoche | 3 | 400 |
| | | FF[4]-IAIK | 1 | |
| | | ER[5]-IAIK | 1 | |
| PP-F | Prime+Probe (PP) Family | PP-IAIK, PP-Jzhang | 2 | 400 |
| S-FR | Spectre-like Variants of FR | Spectre-FR[6]-Kocher, Spectre-FR-Opsxcq, Spectre-FR-Idea4good | 3 | 400 |
| S-PP | Spectre-like Variants of PP | Spectre-PP-Trippel | 1 | 400 |

[1]Samples: cf. [19] for the source of the attack samples. [2]#C: number of collected attacks.[3]#M: number of mutated variants. [4]**FF:** Flush+Flush attack. [5]**ER:** Evict+Reload attack. [6]**Spectre-FR:** Spectre V1 Attack.

### *B. Similarity-based Detection and Classification*

In this subsection, we propose an approach to calculate the similarity between two CST-BBSes. We first show how to calculate the distance between two CSTs, based on which we calculate the complete distance of two CST-BBSes for the similarity comparison.

*1) Distance Between Two CSTs:* Consider two CSTs $\tau_i = S_i \xrightarrow{b_i} S_i'$ for $i = 1, 2$, let $\mathsf{IS}_i$ be the instruction sequence of the BB $b_i$ and $\mathsf{CSP}_i$ be the pair of cache states $(S_i, S_i')$. We measure the similarity between two CSTs from two dimensions, i.e., IS and CSP.

To measure the similarity between two instruction sequences $\mathsf{IS}_1$ and $\mathsf{IS}_2$, we first perform an instruction normalization [20] with following three rules to eliminate the changes introduced by compilers: (1) The immediate data is replaced by "imm". (2) The accessed memory addresses are replaced by "mem". (3) The registers are replaced by "reg". For example, the instruction `mov -0x18(%rbp),%rax` will be normalized as `mov mem, reg`. After normalization, the distance $\mathsf{D}_{\mathsf{IS}_{1,2}}$ between $\mathsf{IS}_1$ and $\mathsf{IS}_2$ is measured via the normalized Levenshtein distance [21], defined by $\mathsf{D}_{\mathsf{IS}_{1,2}} = \frac{\mathsf{LevenshteinDistance}(\mathsf{IS}_1,\mathsf{IS}_2)}{\max(\mathsf{len}(\mathsf{IS}_1),\mathsf{len}(\mathsf{IS}_2))}$.

Now, we measure the similarity between $\mathsf{CSP}_1$ and $\mathsf{CSP}_2$. Recall that $\mathsf{CSP}_i = (S_i, S_i')$, where $S_i = (AO_i, IO_i)$, $S_i' = (AO_i', IO_i')$ for $i = 1, 2$. The distance $\mathsf{D}_{\mathsf{CSP}_{1,2}}$ between $\mathsf{CSP}_1$ and $\mathsf{CSP}_2$ is defined by: $\mathsf{D}_{\mathsf{CSP}_{1,2}} = |P_2 - P_1|$, where $P_i = \frac{|AO_i - AO_i'| + |IO_i - IO_i'|}{2}$ for $i = 1, 2$. Intuitively, $P_i$ measures the cache changes in the CST $\tau_i$. The resulting distance $\mathsf{D}_{\mathsf{CSP}_{1,2}}$ between $\mathsf{CSP}_1$ and $\mathsf{CSP}_2$ measures the similarity of cache changes between the CSTs $\tau_1$ and $\tau_2$.

With $\mathsf{D}_{\mathsf{IS}_{1,2}}$ and $\mathsf{D}_{\mathsf{CSP}_{1,2}}$, the similarity of two CSTs $\tau_1$ and $\tau_2$ is measured by $\mathsf{Distance}(\tau_1, \tau_2) = \frac{\mathsf{D}_{\mathsf{IS}_{1,2}} + \mathsf{D}_{\mathsf{CSP}_{1,2}}}{2}$.

*2) Distance Between Two CST-BBSes:* To measure the similarity degree between two CST-BBSes, we adapt the Dynamic Time Warping (DTW) algorithm [7], which is widely used in attack identification [22]. The main idea of the DTW algorithm is that it uses a distance function to compare the similarity degree between two given subsequences, and match the similar subsequences in two complete sequences in order. In this work, we use $\mathsf{Distance}(\tau_1, \tau_2)$ as the distance function in the DTW to support the similarity comparison of two CST-BBSes. The distance $D$ calculated by the DTW is in the range [0, $\infty$), the larger the distance, the less the similarity. In this paper, we use $\frac{1}{D+1}$ to convert the distance into the range (0,1], as the similarity score, thus, the larger the score, the more the similarity.

*3) Attack Detection and Classification:* To deploy our approach, we build a repository of attack behavior models from the PoCs of existing attacks. Given a target program, SCAGUARD first performs the attack behavior modeling on the target program. Then, it calculates the similarity degree between the target program and all the PoCs of attacks, respectively. The high similarity degree implies that the target program belongs to the same attack family as the compared attack PoC. If all of the similarity scores between the target program and the selected PoCs are lower than a threshold, e.g., $45\%$ (cf. Section V for optimal threshold selection), the target program is considered to be a benign one.

TABLE III: THE BENIGN DATASET

| Type | Description | Number |
|---|---|---|
| SPEC2006 | All of the SPEC2006 test cases in [23] | 12 |
| LeetCode | 230 solutions to Leetcode algorithm problems are collected [6], [24][1] | 230 |
| Encryption | 6 commonly-used crypto-systems such as RSA and AES are collected and mutated. Then randomly select 25 samples from each category [6] | 150 |
| Server Applications | SQLite, OpenSSH, OpenSSL, Vsftpd, Thttpd, Gzip, OpenVPN, OpenNTPD [25] | 8 |

[1] To make the number of benign samples the same as the number of an attack type (i.e., 400), we collect 280 leetcode solutions.

## IV. EVALUATION

### *A. Experimental Setup*

**Platform.** We conduct our experiments under Ubuntu 16.04 running on the PC with Intel i7-6700 CPU and 32 GB RAM.

**Attack Dataset.** As listed in Table II, the attack dataset contains 4 attack types, where FR-F and PP-F are called *source attacks* and their Spectre-like variants are called *Spectre-variants*.

To expand the diversity and number of subjects, similar to the recent works (e.g., [26], [27]), we create 400 PoCs (including the samples) for each attack type via code mutation [28], where the variants of source attacks are called *mutated source attacks* and the variants of Spectre-variants are called *mutated Spectre-variants*. Note that we retain the attack functionality during mutation so that the code mutation does not fundamentally break the attack behaviors.

**Benign Dataset.** The same as each attack type, we collect 400 benign programs which are listed in Table III. Similar to prior works [4]–[6], 392 out of 400 benign programs are the SPEC2006 cases, commonly-used cryptosystem, and diverse Leetcode solutions, which have different degrees of memory accesses. Furthermore, we also collect various real server applications, following the prior work in security community (cf. *Server Applications* in [25]), we choose the 8 most commonly-used real-world applications, such as SQLite, OpenSSH, etc.

### *B. Accuracy of Attack-relevant BB Identification*

In this subsection, we evaluate if SCAGUARD can identify all the attack-relevant BBs by counting #BB, #TAB, #IAB, and #ITAB (cf. Table IV). The results in Table IV indicate that SCAGUARD can achieve an average accuracy of 97.06% (66 out of 68 BBs) in identifying the manually identified attack-relevant BBs and most attack-irrelevant BBs are eliminated.

**Summary:** SCAGUARD can effectively identify attack-relevant BBs and shrink the size of BBs for further analysis.

### *C. Effectiveness of SCAGUARD*

To evaluate the effectiveness of SCAGUARD, we consider five different scenarios, as summarized in Table V. We measure the similarity between Flush+Reload and another Flush+Reload implementation (**in S1**), Flush+Reload and Evict+Reload (**in S2**), Flush+Reload and Prime+Probe (**in S3**), Flush+Reload and its Spectre-variant (**in S4**), Flush+Reload and the benign program (**in S5**), respectively. Details of these scenarios please refer to [25].

The similarity score in Table V shows that with the increase of the differences between the programs, the similarity degrees reported by SCAGUARD gradually decrease. Also, the similarity degrees for all the attacker-only scenarios are greater than $66\%$ while the similarity degree between the attack and the benign program is less than $16\%$. It indicates that SCAGUARD can effectively distinguish between attack programs and their variants, as well as benign programs.

**Summary:** SCAGUARD is very effective for detecting cache side-channel attacks.

TABLE IV: Results of Attack-relevant BB Identification

| Attack | #BB[1] | #TAB[2] | #IAB[3] | #ITAB[4] | Accuracy |
|---|---|---|---|---|---|
| FR-F | 18174 | 98 | 3612 | 95 | 96.94% |
| PP-F | 1547 | 40 | 914 | 39 | 97.50% |
| S-FR | 1334 | 64 | 352 | 62 | 96.88% |
| S-PP | 1572 | 70 | 872 | 69 | 98.57% |
| **Avg.** | **5657** | **68** | **1438** | **66** | **97.06%** |

[1]*#BB*: number of BBs. [2]*#TAB*: number of manually identified attack-relevant BBs (ground-truth). [3]*#IAB*: number of identified attack-relevant BBs by our approach. [4]*#ITAB*: number of manually identified attack-relevant BBs that are identified by our approach.

### *D. Comparison with Prior Approaches*

In this subsection, we compare SCAGuard with the rule-based detection approach SCADET [6] and machine learning-based approaches with different classifiers: Support Vector Machine based one of NIGHTs-WATCH (SVM-NW) [5], Linear Regression based of NIGHTs-WATCH (LR-NW) [5] and K-Nearest Neighbors Algorithm based malicious loop finding approach (KNN-MLFM) [4]. We note that SCADET is the unique learning-free approach, while the others are the most highly cited papers published within the past 5 years using machine learning and have been proven to be very effective in detecting Flush+Reload and Prime+Probe attacks [4], [5], [22]. For a fair comparison, SCADET is the author-provided tool [6] and we reproduce SVM-NW, LR-NW, and KNN-MLFM to their best performance according to their papers. We conduct four types of evaluation E1∼E4, where SCAGuard uses only one PoC for each attack type for attack behavior modeling and the three learning-based approaches use 10-fold cross validation to obtain the best model with the fine-tuned parameters. Besides, SCADET always uses its designated rules for each evaluation. The details of samples chosen for training/modeling or classification refer to [25].

- **E1: Classification of mutated-variants**. The mutated-variant classification task is to classify mutated variants (i.e., FR-F, PP-F, S-FR and S-PP) when some of them are known to the defender.
- **E2: Classification of Spectre-like variants**. This classification task is to classify spectre-like variants (i.e., S-FR and S-PP) when only their non-spectre-like counterparts (i.e., FR-F and PP-F) are known to the defender.
- **E3: Classification of other attack family's variants (Generalizability)**. To evaluate the generalizability of SCAGuard, we consider two sub-tasks. The first one is to classify Prime+Probe Family when only the Flush+Reload Family is known to the defender. The second one is to classify Flush+Reload Family when only Prime+Probe Family is known to the defender.
- **E4: Classification of obfuscated variants (Robustness)**. To evaluate the robustness of SCAGuard against the attacker who tries to obfuscate an existing PoC in order to bypass the detection approach, for each PoC out of 400 PoCs of the attack type FR-F (resp. PP-F), we generate obfuscated variants by applying the commonly-used obfuscation technique, polymorphic technique [29], resulting $400 \times 2$ new obfuscated variants. These obfuscated variants inserted with junk code (e.g., NOP) have, on average, 70.49% more BBs per sample than the original one. Our goal is to detect the obfuscated variants while only their non-obfuscated counterparts are known to the defender.

**Results Analysis**. The results are reported in Table VI, where the best ones are highlighted in **bold font**. We can observe that SCAGuard is very effective for all the tasks E1∼E4. Its precision is 3.25-70.27% higher than the three learning-based approaches with higher Recall and F1-score. SCAGuard also outperforms the learning-free tool SCADET. In particular, for E2∼E4, the learning-free tool SCADET fails to detect any of variants, indicating that our attack behavior models are better than the manually designed rules of SCADET.

TABLE V: similarity comparison of 5 typical scenarios

| No. | Scenario | Description | Score |
|---|---|---|---|
| S1 | Flush+Reload (FR) / Another implementation | Different implementations of the same attack | 94.31% |
| S2 | Flush+Reload / Evict+Reload (ER) | Different variants of the same attack | 84.32% |
| S3 | Flush+Reload / Prime+Probe (PP) | Different attacks exploiting the same vulnerability | 74.48% |
| S4 | Flush+Reload / Its Spectre variant | Different variants exploiting different vulnerabilities | 66.92% |
| S5 | Flush+Reload / Benign Program | An attack program and a benign program | 15.10% |

Note that in the two sub-tasks of E3 (denoted by E3-1 and E3-2 in Table VI), we can observe that the precision of all the three learning-based approaches drops dramatically, indicating that learning-based approaches without a large dataset of high-quality training samples such as CSCAs are over-fitted, which greatly reduces their ability to identify and classify attack variants. In contrast, SCAGuard can still achieve the precision of 91.28% and 92.55% in both sub-tasks, respectively, significantly outperforming all the other approaches. These results indicate that SCAGuard is more generalizable, because our approach is not tailored to specific patterns, but a generic design for detecting CSCAs. Recall that CSCAs exploit the timing difference caused by cache operations (e.g., cache hit and cache miss). To probe the time difference, the attacker inevitably needs to perform cache operations multiple times, for attack preparation and attack execution, which definitely changes the cache states. Therefore, even if new cache side-channel attack families appear, our approach SCAGuard can still quickly and automatically build the attack behavior models for them by leveraging the static CFG, HPC data, and cache state changes, then identify such new attack families.

**Summary:** SCAGuard is more effective than prior approaches, in particular on new (Spectre-like, other attack family's, and obfuscated) variants.

## V. Discussion

**Time cost.** In our evaluation, we also record the time costs of attack detection. The average time cost of SCAGuard is 636.96s, comparable to the rule-based method SCADET which is 562.76 seconds. The learning-based methods take 5.91s, 5.66s and 7.20s, respectively. The difference in the time costs is reasonable, as both SCAGuard and SCADET do not have pre-trained models but have to collect runtime information. Thus, similar to SCADET, SCAGuard is more suitable for offline detection scenarios. For instance, SCAGuard can be deployed at the server cluster as a guard for cache attacks. When an untrust program needs to be installed on a server, one can first perform a security check by applying SCAGuard.

Through further analysis, we observe that 56.6% of the time cost is spent on collecting the accessed memory addresses and 39.3% on the file I/O. One potential solution to these time costs is to integrate SCAGuard into kernels or implement it as a hardware module. We leave this optimization as future work.

**Threshold.** The threshold of similarity degree can be used to control the trade-off between false positive and false negative rates. In our experiments, we follow recent works in security community, e.g., HOLMES [30], to choose the optimal one by measuring the Precision, Recall, and F1-Score. The results are shown in Fig.5. We can find that when the threshold falls in 30%∼60%, the corresponding Precision, Recall, and F1-Score are all greater than 90%, so 30%∼60% is an acceptable threshold range. Thus, we use the middle value of 30%∼60%, i.e. 45%, as the threshold for all the other experiments.

**Limitation.** Some attack programs under disguise may need complex input to trigger their hidden malicious behaviors. In this paper, we focus on the programs whose attack behaviors can be triggered

TABLE VI: The classification results of SCAGuard and other 4 existing attack detection approaches

| Approach | E1: Mutated-variants | | | E2: Spectre-like variants | | | E3-1: PP-F | | | E3-2: FR-F | | | E4: Obfuscated variants | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1-score | Precision | Recall | F1-score | Precision | Recall | F1-score | Precision | Recall | F1-score | Precision | Recall | F1-score |
| SVM-NW | 94.58% | 94.20% | 94.24% | 90.49% | 90.0% | 90.04% | 21.01% | 36.25% | 26.61% | 78.99% | 73.75% | 72.51% | 89.49% | 88.89% | 88.88% |
| LR-NW | 68.15% | 51.51% | 49.00% | 66.96% | 55.83% | 52.56% | 75.64% | 72.50% | 71.63% | 64.88% | 63.75% | 63.05% | 42.82% | 64.17% | 51.33% |
| KNN-MLFM | 91.32% | 91.70% | 91.45% | 42.66% | 63.33% | 50.94% | 67.58% | 66.25% | 65.60% | 82.74% | 77.50% | 76.56% | 88.66% | 88.34% | 88.23% |
| SCADET | 50.00% | 27.50% | 35.48% | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SCAGuard | **96.64%** | **96.50%** | **96.52%** | **95.2%** | **95.0%** | **95.03%** | **91.28%** | **91.25%** | **91.25%** | **92.55%** | **91.25%** | **91.18%** | **92.74%** | **92.23%** | **92.25%** |

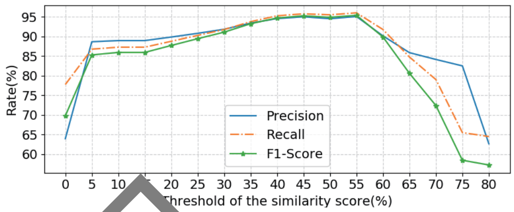



Fig. 5: Classification results of SCAGuard by varying the threshold value.

by directly executing them. Therefore, if the attack behaviors of the target program are not triggered during the data collection, it would not be able to construct a precise attack behavior model by analyzing the dynamic runtime information. Unfortunately, to the best of our knowledge, the existing cache attack detection solutions do not consider such attack scenarios as well [4], [5], [22]. To cope with this issue, one may adapt coverage-driven testcase generation approaches to trigger attack behaviors. We leave this as future work.

## VI. Related Work

To mitigate cache side-channel attacks, novel secure cache architectures are proposed to avoid the malicious eviction of victim's cache lines, and constant-time analysis techniques are proposed to detect or eliminate cache side-channels of programs [3]. Though promising, they are hard to update quickly. To remedy these problems, machine learning-based and rule-based detection approaches are proposed.

**Learning-based detection approaches.** Attack-oriented learning-based approaches, e.g., [31], collect the runtime information of attack programs, based on which, a classifier is trained to detect attacks.

Recent works proposed victim-oriented approaches, e.g., anomaly-based detection method [32], which learns a classifier using the HPC data of benign programs, thus does not require any attack samples. However, the data from a single source may lead to a high false positive ratio and the identified attacks cannot be further classified.

To reduce false positives and classify attacks, Mushtaq et al. [5] proposed to collect the victim's data in two scenarios, i.e., with and without the attacker running in the environment. Wang *et al.* proposed Phased-Guard [22] to identify and classify a given program in two steps. After utilizing the anomaly detection mechanism to check if the victim is being attacked, a multi-class classifier is trained to classify the attack. However, they still require a large number of attack samples and often either fail to detect new attack variants that are not included in the training set or produce false positives. Instead, our approach requires only a few PoCs of existing attacks and is able to detect and classify new variants.

**Rule-based detection approach.** The learning-free approach of SCADET [6] manually extracts cache set access patterns of existing attack programs from which detection rules are created. The heuristic rule based approach relies on manually designed cache access patterns which are labor-intensive. Moreover, the heuristic rules are also not flexible and can be easily bypassed by attack variants. In contrast, our approach automatically builds attack behavior models from PoCs and the model is able to detect and classify new attack variants.

## VII. Conclusion

We proposed a novel approach to detect and classify cache side-channel attacks. We introduced the notion of CST-BBS as attack behavior models which is able to capture both attack-relevant syntactic code information and semantic cache information. We presented an approach to automatically build attack behavior models from PoCs of existing attacks and a similarity comparison approach for detecting and classifying attack variants via attack behavior models. We conducted extensive experiments on various attack and benign programs. The experimental results demonstrate that the proposed approach outperforms, in particular, on new attack variants, existing promising learn-based and rule-based approaches.

## References


[1] Y. Yarom *et al.*, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.
[2] E. Tromer *et al.*, "Efficient cache attacks on aes, and countermeasures," *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, 2010.
[3] D. Ojha *et al.*, "Timecache: Using time to eliminate cache side channels when sharing software," in *ISCA*, 2021.
[4] Z. Allaf *et al.*, "A comparison study on flush+reload and prime+probe attacks on AES using machine learning approaches," in *UKCI*, 2017.
[5] M. Mushtaq *et al.*, "Nights-watch: a cache-based side-channel intrusion detector using hardware performance counters," in *HASP*, 2018.
[6] M. Sabbagh *et al.*, "SCADET: a side-channel attack detection tool for tracking prime+probe," in *ICCAD*, 2018.
[7] D. J. Berndt *et al.*, "Using dynamic time warping to find patterns in time series," in *AAAI Workshop*, 1994.
[8] S. Das *et al.*, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *S&P*, 2019.
[9] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.
[10] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *S&P*, 2019.
[11] "Flushu+reload," https://github.com/SCAGuard/RunningExample.
[12] D. Gruss *et al.*, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security*, 2015.
[13] D. Gruss *et al.*, "Flush+flush: A fast and stealthy cache attack," in *DIMVA*, 2016.
[14] "Angr binary analysis tool," http://angr.io/.
[15] "Intel processor trace within perf tools," https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html.
[16] "Intel processor trace," https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html.
[17] "Maximum spanning tree using prim's algorithm," https://www.geeksforgeeks.org/maximum-spanning-tree-using-prims-algorithm/.
[18] "A cache simulator," https://github.com/jiangxincode/CacheSim.
[19] "The attack dataset," https://github.com/SCAGuard/DataSet.
[20] Z. Xu *et al.*, "SPAIN: security patch analysis for binaries towards understanding the pain and pills," in *ICSE*, 2017.
[21] Y. Xu *et al.*, "Patch based vulnerability matching for binary programs," in *ISSTA*, 2020.
[22] H. Wang *et al.*, "Phased-guard: Multi-phase machine learning framework for detection and identification of zero-day microarchitectural side-channel attacks," in *ICCD*, 2020.
[23] J. Nomani *et al.*, "Predicting program phases and defending against side-channel attacks using hardware performance counters," in *HASP*, 2015.
[24] "Leetcode: an online programming platform," https://leetcode.com.
[25] "More details for scaguard," https://github.com/SCAGuard.
[26] M. C. Tol *et al.*, "Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings," in *EuroS&P*, 2021.
[27] A. Alsaheel *et al.*, "ATLAS: A sequence-based learning approach for attack investigation," in *USENIX Security*, 2021.
[28] "Mutate_cpp," https://github.com/nlohmann/mutate_cpp.
[29] "Polymorph-lib," https://github.com/JarateKing/polymorph-lib.
[30] S. M. Milajerdi *et al.*, "HOLMES: real-time APT detection through correlation of suspicious information flows," in *S&P*, 2019.
[31] J. Demme *et al.*, "On the feasibility of online malware detection with performance counters," in *ISCA*, 2013.
[32] M. Chiappetta *et al.*, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.