Locality Based Cache Side-channel Attack Detection*

Limin Wang[1], Lei Bu[1] (✉), and Fu Song[2]

[1] State Key Laboratory of Novel Software Techniques
Nanjing University, Nanjing, Jiangsu 210023, China
wanglimin@smail.nju.edu.cn bulei@nju.edu.cn
[2] School of Information Science and Technology
ShanghaiTech University, Shanghai 201210, China
songfu@shanghaitech.edu.cn

Abstract

Cryptographic algorithms are fundamental to security. However, it has been shown that secret information could be effectively extracted through monitoring and analyzing the cache side-channel information (i.e., hit and miss) of cryptographic implementations. To mitigate such attacks, a large number of detection-based defenses have been proposed. To the best of our knowledge, almost all of them are achieved by collecting and analyzing hardware performance counter (HPC) data. But these low-level HPC data usually lacks semantic information and is easy to be interfered, which makes it difficult to determine the attack type by analyzing the HPC information only.

Actually, the behavior of a cache attack is *localized*. In certain attack-related steps, the data accesses of cache memory blocks are intensive, while such behavior can be distributed sparsely among different attack steps. Based on this observation, in this paper, we propose the *locality*-based cache side-channel attack detection method, which combines the low-level HPC running data with the high-level control flow graph (CFG) of the program to achieve locality-guided attack pattern extraction. Then we can use GNN graph classification technology to learn such attack pattern and detect malicious attack programs. The experiments with a corpus of 1200 benchmarks show that our approach can achieve 99.44% accuracy and 99.47% F1-Score with a low performance overhead.

1 Introduction

The past ten years have seen increasingly rapid exploits in the field of cache side-channel attacks, such as Flush+Reload [1], Flush+Flush [2], Prime+Probe [3], and etc. This type of attack is extremely harmful. On the one hand, cache attacks themselves can be used as a standalone attack against cryptographic algorithms. On the other hand, they can also be combined with attacks that exploit hardware vulnerabilities like Meltdown [4] and Spectre [5] to form stronger attacks.

To defend against the above-mentioned attacks, many defense techniques have been proposed to mitigate vulnerabilities, such as SP cache and PL cache and etc [6, 7, 8, 9, 10, 11, 12, 13]. However, as the number of exploits and varieties are gradually increased, some of these defenses are obsolete and can be bypassed. As an effective complement to the mitigations against vulnerabilities, attack detection technology has played an important role in system security.

The cache attack detection technology identifies the malicious intrusions by collecting and analyzing the runtime signals of target programs such as hardware performance counters (HPC). In addition, the signals commonly used for attack detection usually include instruction execution

*This work is supported by the Leading-edge Technology Prohgram of Jiangsu Natural Science Fundation (No. BK20202001), and the National Natural Science Foundation of China (NSFC) under Grants (No. 62072309).

traces, memory access traces, and so on. These attack detection methods driven by runtime signal analysis can be called as detection-based defenses [14]

Detection based methods usually collect the necessary low-level information of hardware performance events and use the machine learning method for attack identification and classification. As the most widely used detection-based solution, the attack-oriented detection strategy needs to train the HPC data of attack samples and benign samples to classify the target program [15, 16, 17, 18, 19, 20].

Existing works on the attack-oriented detection only consider the usage pattern of cache. However, cache is a shared component. The usage pattern of cache can be easily interfered by other normal cache accesses, which will lead to inaccurate recognition. Furthermore, hardware-level runtime information contains few semantics information of the attack program. Therefore, it is difficult to identify different types of attacks by using the low-level cache pattern only, which may cause false alarms.

Actually, the behaviors of cache attacks are normally distributed sparsely in different steps of the attack program's behavior. Compared with the behavior of the program's normal step, there always exist a large number of attack-related program controls and data dependencies in the program's attack steps. We call this phenomenon as *Locality* in this paper.

Based on this observation, to overcome the above-mentioned challenges, we propose the locality-based cache side-channel attack detection method with a novel program representation, which bridges the gap between the low-level runtime information and the high-level static information of the target program.

The key idea of our method is to reduce the impact of runtime noise on pattern extraction by matching the attack-related program behavior in the static level, i.e., the control flow praph (CFG) of the program, with the dynamic level, i.e., the runtime HPC data, to increase the accuracy of attack recognition and decrease the false alarm rate.

In order to integrate the runtime HPC data and the static CFG, we first find and label the attack-related program structures in the CFG. Then, with the CFG as the backbone, we encode different types of edges and the corresponding HPC data triggered by the instruction behavior in the control flow nodes into the graph with heterogeneous edges. The obtained control flow graphs will eventually be trained with Graph Neural Networks (GNN). Finally, the trained GNN graph classifier can be used for the identification and classification of potential attack programs.

The summary of the contributions of this work is as follows:

1. We propose a new program representation to highlight the attack behavior of a program. By integrating the runtime HPC data into the corresponding control flow graph of a program, the attack-related program controls and data dependencies are highlighted.

2. With the novel program representation that combines dynamic data and static data according to the attack locality, our method uses GNN to identify the target program accurately with a low performance overhead.

3. We implement the attack detection and classification framework with a data set contains *Flush+Reload* (400 samples), *Prime+Probe* (400 samples), and *benign applications* (400 samples). The classification results show our approach brings an average 99.44% accuracy and 99.47% F1-Score.

2 Background

2.1 Control flow graphs

A control flow graph (CFG) of a program represents the structure and all the potential paths of a program. Therefore, CFG has been widely used in attack detection [21, 22].

Definition 1 The *CFG* of a program is a tuple, that $CFG = (Vertices, Edges)$.

- *Vertices* is a set of vertices, each vertice consists of an *Instruction* and the corresponding *Instruction Address*.
- *Edges* is a set of control flow edges which connecting two vertices.

There are mainly three typical subgraph structures in a CFG of a program. Figure 1 (a) shows the subgraph with a sequence of basic blocks, in which all the blocks will be executed one by one. Figure 1 (b) shows the subgraph with a branch structure, which represents conditional branches like *if-else*, and *switch*. The subgraph with a circle indicates the program loop. The program can repeatedly execute the basic blocks in the loop body (shown in Figure 1 (c)).

2.2 Hardware performance counter

Hardware performance counters (HPC) are a number of registers equipped in modern microprocessors to store data of various CPU-related events, e.g. clock cycles, cache hits, etc. Existing works [15, 16, 17, 18, 19, 20] usually use sampling technology to collect the number of occurrences of HPC events in a specified time interval. Then, the execution pattern in the collected data is extracted to identify potential attack programs.

The collected HPC-related data usually contains the current time *Timestamp* and the HPC event information *HPC_Event*, which is denoted as *HPC_Data* and shown in *Definition 2*.

Definition 2 The *HPC_Data* of a program is a tuple, $HPC\_Data = (Timestamp, HPCEvent)$.

- *Timestamp*: Each $t_i \in Timestamp$ represents the timestamp when the data was collected.
- *HPCEvent*: Each $e_i \in HPCEvent$ represents the HPC data collected at time spot t_i[1].

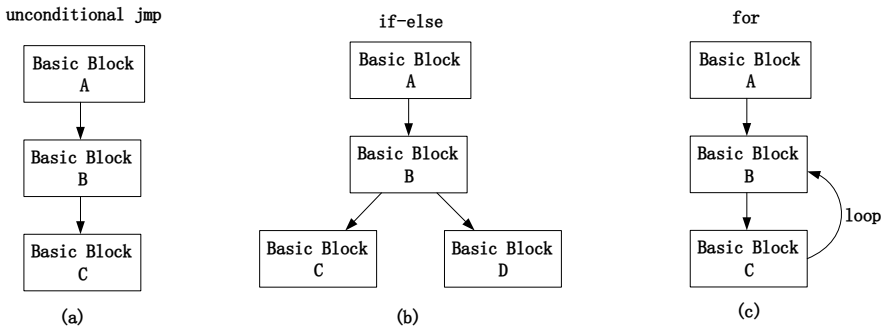

Figure 1: The typical subgraph structures of control flow graphs

[1]Noted that e_i is a tuple of HPC event information, including number of occurrences of cache misses, cache hits, branch misses and etc. Due to space limitation, we do not give the complete list of e_i in the definition [23].

2.3 Cache side-channel attacks

As the cache side-channel attacks are widely concerned, different exploits emerge endlessly. Two of the most concerned attacks are *Flush+Reload* [2], and *Prime+Probe* [3]. Both of them are access-based cache attacks, and can exploit Last Level Cache (LLC) side-channel vulnerability. More importantly, they can be combined with the recently proposed hardware vulnerabilities [4, 5].

Flush+Reload [2] is a class of attack which needs shared memory between the attacker and the victim. Its high accuracy and ease of operation make it extremely threatening. In Figure 2, each rectangle represents a memory block, and the blue area represents the memory blocks shared by the attacker and the victim. The Figure 2 (a) shows that the instruction *clflush* can be used directly to invalid the target cache lines. Then the attacker needs to flush the target cache lines repeatedly and wait for the victim's execution (shown in Figure 2 (b)). When the attacker accesses the flushed cache lines again, if the required time is short, it means *cache hit* happens and the victim's sensitive information has been cached before. This entire attack is shown step by step in Figure 2.

However, not all of the instruction set architectures (ISA) have the *clflush* instruction. Thus the *Evict* strategy is proposed to replace the *Flush* operation in *Flush+Reload* attack. More specifically, attackers are able to repeatedly read and write the specified cache sets. If all the cache lines in a cache set are not empty, the oldest one will be evicted according to the cache replacement policy. Therefore, the victim's cache lines will be replaced by the newly stored data. Then, similar to *Flush+Reload* attack, the attacker needs to wait for the victim to execute and probe the cache lines again. Finally, the secret information can also be leaked by analyzing the access time.

Prime+Probe [3] is another widely focused attack with fewer restrictions. It does not rely on shared space nor special instructions. *Prime+Probe* accesses memory blocks selected (the blue rectangles shown in Figure 3 (a)) to fill the target cache sets and monitor these cache lines. Similar to *Flush+Reload*, *Prime+Probe* needs to repeat the memory access operations and wait for the victim's execution (shown in Figure 3 (b)). When the victim executes the critical operation, the attacker will access the selected memory blocks again (shown in Figure 3 (c)). If the access time of certain cache line is significantly longer than the access time of other cache lines, it means that this cache line has been accessed by the victim. In this way, the victim's cache use pattern can be observed by the attacker.

In 2018, hardware vulnerabilities are reported to be combined with existing cache side-channel attacks to form new threatening attacks. In Figure 4, Meltdown [4] and Spectre [5]

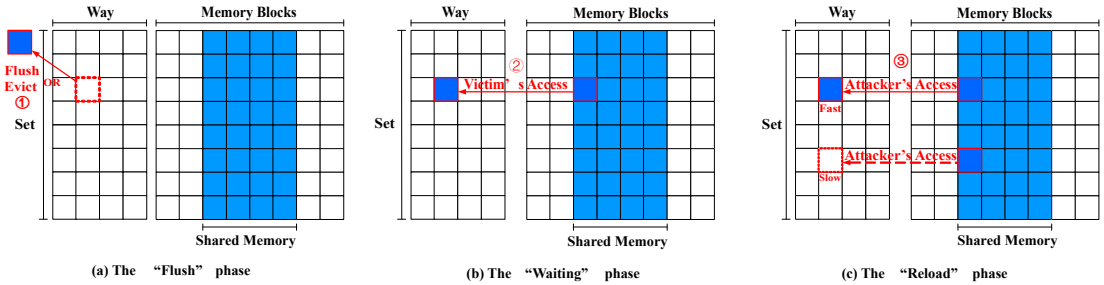

Figure 2: The Flush+Reload Attack

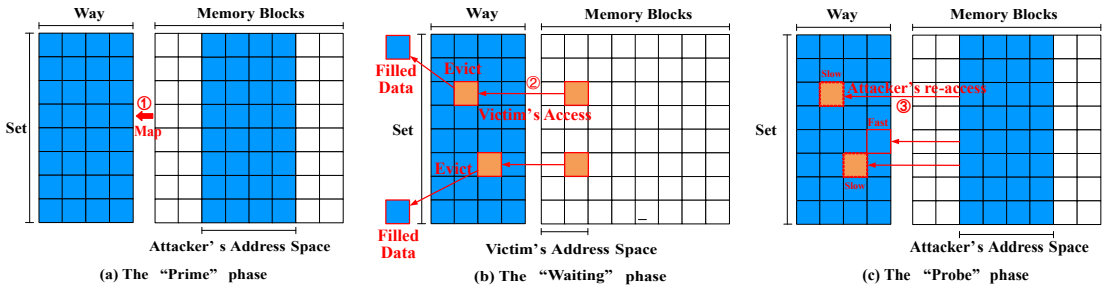

Figure 3: The Prime+Probe Attack

exploit out-of-order and branch prediction respectively to perform transient execution attacks. During the attack, they train a *Speculative Execution*, and illegally leak the secret-related data into the cache. When the CPU finds the wrong speculation, the results of speculative execution will be discarded. However, the sensitive information is still in cache. Then, such information can be retrieved by attackers easily.

2.4 Attacker-oriented cache side-channel attack detection

Cache side-channel attack detection technology can be divided into two major categories: attack-oriented detection and victim-oriented detection [24, 25]. This paper focuses on the attacker-oriented detection technology. The key point of attacker-oriented detection is to extract the attack pattern from known attacks and perform pattern matching with target programs. Chiappetta et al. [18] believe that it is possible to continuously compute the correlation of HPC values between a target program and the attack program. If the correlation is larger than a certain predetermined threshold, the given program could be considered as an attack program. In addition, they use the machine learning method to train HPC models for cache attacks, and then identify whether the HPC value generated by the target program matches the previously trained model. Furthermore, supervised learning is also introduced to train attacker and victim programs. Then, the trained classifier can be used to identify whether the given program is a benign program or a side-channel attack program. Similarly, in [15, 16, 17], they

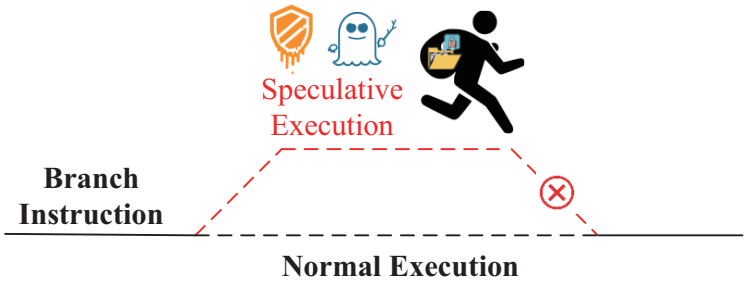

Figure 4: The Speculative Execution Attack

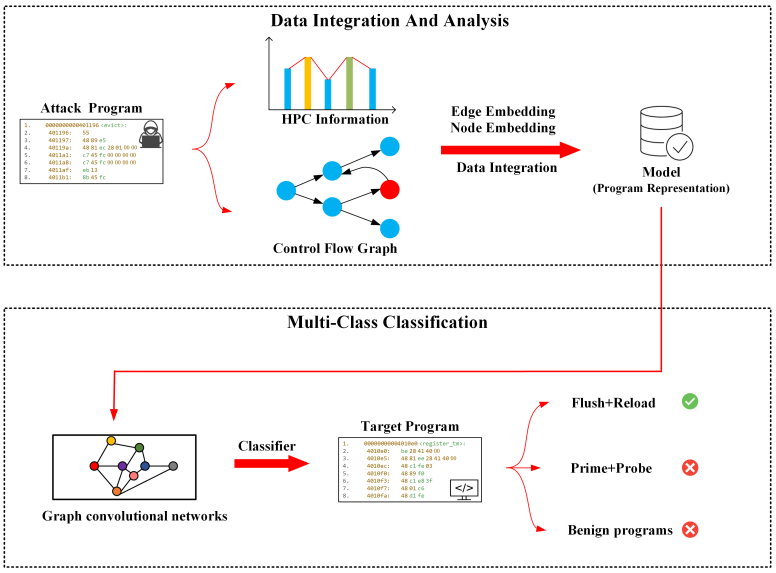

Figure 5: The workflow of the cache side-channel attacks detection

also collect HPC data and use machine learning to identify cache attacks.

By reproducing existing works and manually analyzing the collected HPC data, we observe two major problems may caused by existing HPC value based detection methods:

1. On the one hand, when the attack program is running, the corresponding HPC value will be relatively high. But while other normal codes are running, the HPC value may also be high, which will lead to high false alarm rate in the existing HPC data pattern extraction and attack identification method.

2. On the other hand, different side-channel attacks use different attack operations to clear the target cache lines. For example, *Flush+Reload* invalidates the cache lines through the *clflush* instruction, while *Prime+Probe* evicts cache lines by accessing the cache repeatedly. All of these different attack operations may change the HPC value. Therefore, if we use HPC pattern as the detection metric only, it will be difficult to distinguish these attack operations in different attacks.

3 Locality based cache side-channel attack detection

In this section, we introduce the locality phenomenon of attack steps in the cache side-channel attacks. Then, based on this observation, our specifically designed locality based attack detection is illustrated in detail. As Figure 5 shows, the proposed method in this paper can be divided into two steps. The first phase is *Data Integration*. In this phase, we collect the runtime HPC information of the target program and embed the HPC data into the basic blocks of the CFG of the target program according to the locality of attack. The second phase is *Multi-Class Classification*. In this phase, the dataset collected in the previous stage will be fed to the GNN based graph learning and classification. Then, the GNN based classification can tell whether the target program is benign or falls into an attack class, i.e., *Flush+Reload* or *Prime+Probe*.

3.1 Locality of the cache side-channel attack program

As we all know, the CFG of a program consists of series of basic blocks. We observed that, given the CFG of an attack program, there exist several groups of connected basic blocks in the CFG. The instructions in these groups of blocks frequently perform operations on certain memory blocks. Furthermore, these frequently visited memory blocks can be mapped back to the victim's cache lines. Different from the instructions in the above-mentioned group of blocks, the instructions in other basic blocks have much fewer access to the victim-related data. The above-mentioned behavior is called *Locality* in our paper. We call such group of blocks which are attack-intensive as an *Attack Step*.

Take the attack *Flush+Reload with Spectre* for example, the *Flush+Reload with Spectre* is a widely concerned new variant of the *Flush+Reload* attack. We demonstrate the *locality* phenomenon in detail as follows. Figure 6 shows the simplified control flow graph of the program *Flush+Reload with Spectre* with special emphasis of three attack step-related basic blocks[2]. The code with respect to these three steps are given in Algorithm 1, Algorithm 2, and Algorithm 3 correspondingly.

Let us look at the CFG of the attack program first. In the *Flush* phase, there exists a circle in the CFG. This group of basic blocks represents the loop in Algorithm 1. In the *Spectre* phase, the grouped nodes 21, 22, and 23 form a tree structure, which represents the *if-else* structure in Algorithm 2. Similarly, the basic blocks of *Reload* phase are also grouped and form a tree structure with a circuit, which represent the loop and the *if-else* structure in Algorithm 3.

Now, we observe the algorithms in detail to analyze the data dependency. The first attack step is *Flush* shown in Algorithm 1. In this step, the attacker frequently flushes the *array2* to invalid the cache lines mapped from the items in *array2*. The *Spectre attack* step (Shown in Algorithm 2) tries to mapped the victim's data to cache lines by illegally accessing the *array2* items that contains *secret_data* during the speculative execution. The last attack step *Reload* (Shown in Algorithm 3) probes each item in *array2* and finds the one that contains *secret_data* by analyzing the access time.

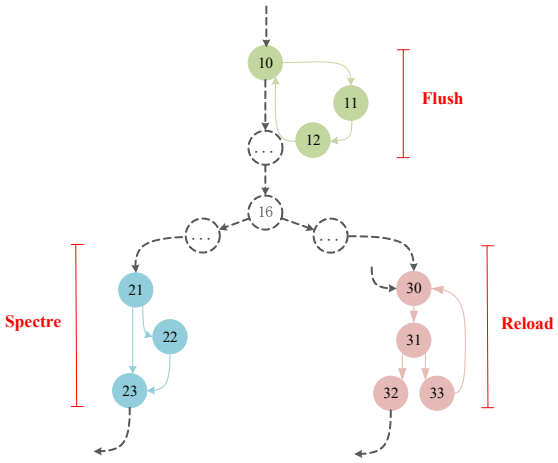

Figure 6: The attack steps in the *Flush+Reload with Spectre attack*

[2]The complete control flow graph is too complex and unreadable. In order to facilitate the introduction of attack locality, the control flow structure that has nothing to do with the attack is represented by a dotted line in the simplified control flow graph.

Algorithm 1 The Key Code In Flush Phase

Input: The attacker-selected memory blocks *array2*;

```
1: /* Attack Step 1: Flush*/
2: for i=0 to 255 do
3:    clflush(array2[i])
4: end for
```

Algorithm 2 The Key Code In Spectre Attack Phase

Input: The attacker-selected memory blocks *array2*, The index x;

```
1: /* if given x>array1_size, array1 is a base address, x is the offset,
   and secret_data=*(array1+x)=array1[x].*/
2: if  x <array1_size then
3:    tmp_data &= array2[array1[x]];
4: end if
```

Algorithm 3 The Key Code In Reload Phase

Input: The attacker-selected memory blocks *array2*;
Output: The leaked data *secret_data*;

```
1: /* Attack Step 3: Reload */
2: for i=0 to 255 do
3:    time1 = RDTSCP;
4:    Read array2[i];
5:    time2 = RDSTCP;
6:    if time2 - time1 <CACHE_HIT_THRESHOLD then
7:       secret_data=CHAR(i);
8:       return  secret_data;
9:    end if
10: end for
```

As shown in the above example, the attack-related basic blocks are often distributed in the control flow graph according to the attack steps, and all of these basic blocks within the same attack steps conduct intensive operations on the same memory blocks. This example shows the locality phenomenon proposed in this paper.

As described in section 2.4, we find the collected HPC data can be easily interfered by other normal behaviours. If we use HPC data only to extract the pattern and identify the attack, the detection result will be affected by the runtime noises. Therefore, in the next paragraph, we propose a data embedding method for program representation. By integrating the dynamic HPC data and static CFG data as the program representation to highlight the attack locality, we can reduce the impact of runtime noise and improve the attack detection accuracy.

3.2 Locality guided program representation generation

In this subsection, we introduce the locality guided data embedding methods for program representation generation. Intuitively speaking, we map the attack-related running HPC data with the CFG of the target program to highlight the attack-related behavior and structure in the program for future detection.

To integrate HPC data into the CFG of a target program, we need to find the common items that can connect these two parts. From the definition given in section 2, a CFG consist

Table 1: Selected Events related to cache side channel attacks

Scope	Description	Event
L1 Cache	L1 Data Cache Load Miss	L1-DCLM
	L1 Data Cache Load Hit	L1-DCLH
	L1 Data Cache Store Hit	L1-DCSH
	L1 Instruction Cache Load Miss	L1-ICLM
LLC Cache	LLC Load Miss	LLC-LM
	LLC Load Hit	LLC-LH
	LLC Store Miss	LLC-SM
	LLC Store Hit	LLC-SH
Others	Branch Miss	OTH-BM
	Branch Load Miss	OTH-BLM
	Cache Miss	OTH-CM

of blocks with instructions information and control flow edges, while the traditional HPC data includes only timestamp and HPC events. Thus, there is no explicit common item between *CFG* and *HPC_Data*.

In order to connect *HPC_Data* with *CFG*, we propose to extend the scope of the data recorded in *HPC_Data* and record the corresponding instruction address as well. More specifically, when tracking the execution of instructions in a program, the currently executed instruction and its address can be recorded. Meanwhile, HPC information is also collected (Noted that the HPC events needed in this paper and their description is shown in Table 1). In this way, the HPC event and the corresponding instruction address are collected and recorded in the log file. With the instruction address, we can map the dynamic HPC data into the static CFG.

Figure 7 shows an example of the integration of HPC data with the CFG in our running example[3]. This statistical chart shows the HPC value of each basic block. The horizontal coordinate represents the index of each basic block, and the vertical coordinate shows the HPC value. The flattened control flow graph below the graph is transformed from Figure 6. In this flattened graph, the basic blocks are arranged in chronological order of program execution. Figure 7 shows that if we can map the HPC value with the three attack steps, the dynamic HPC noise, e.g. HPC data on CFG nodes 27 which is not attack-related, can be recognized. Therefore, what we need to do is to find a method that can detect such attack steps with HPC data patterns efficiently.

To reduce the impact of dynamic noise, we propose a novel program representation, HPC-embedded CFG, that combines the dynamic HPC data and the static CFG to highlight the locality of the attack program.

Definition 3 The HPC-embedded CFG G of a program is a tuple, that $G = (V, E, A, C)$.

- V is a set of nodes, and each node $v_i \in V$ represents the basic block of control flow.
- E is the set of control flow edges.

[3]Noted that each basic block corresponds to 11 HPC events. Due to space limitation, we simplify the graph by accumulating the values of 11 events.

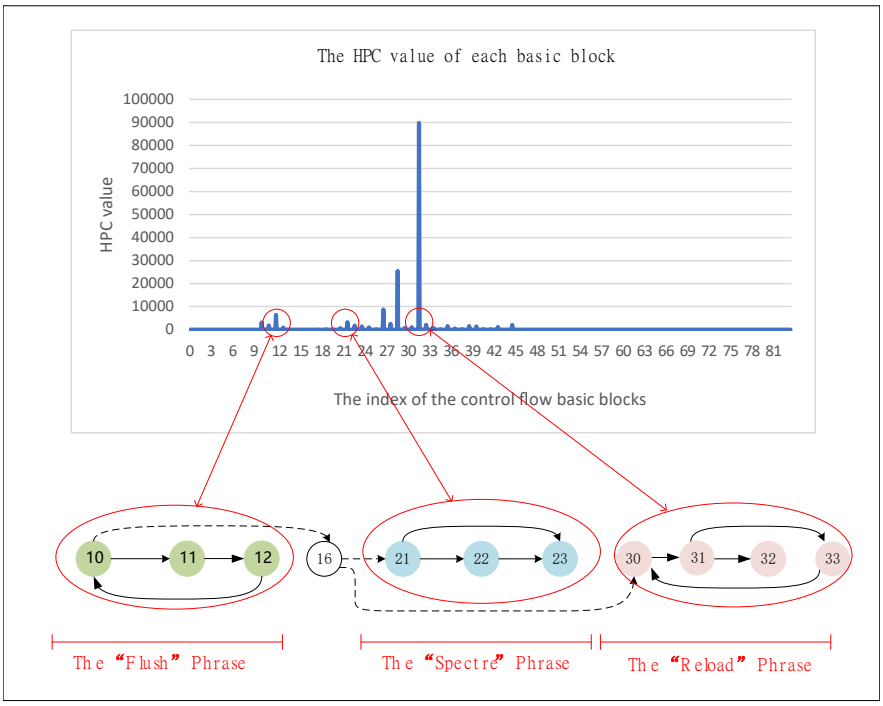

Figure 7: The mapping relationship between dynamic runtime information and static control flow graph

- A represents the features of V, and the feature of each node v_i is denoted as *A[i]*. *A[i]* is the collected HPC data that mapped to the node v_i.
- C contains the types of each CFG edge, and each edge *(i,j)* is labeled as *C[i.j]*.

Given a target program, the dynamic running HPC data and the static CFG of the program, it is obvious that the node set V and edge set E in the HPC-embedded CFG G are directly from the *Vertices* and *Edges* in its CFG. Therefore, we focus on the generation of the components A and C in the following.

According to the phenomenon of attack locality, the instructions in the attack related basic blocks within the same attack step frequently access certain memory blocks, which makes the HPC data within certain blocks varies. To embed these dynamic HPC features into the graph, every node $v_i \in V$ is appended with a vector that contains 11 HPC items, shown in Table 1, as its node feature (*A[i]*).

The locality phenomenon also suggests that the attack related basic blocks within the same attack step are grouped and have unique structures. Therefore, in the HPC-embedded CFG G, we also have a component C to indicate the CFG structure information on edges. Firstly, we list several class of structures according to the characteristics of cache attacks as follows.

1. type 0: The circuit structure represents a *loop*. Loops are important used in attack program, because cache attacks need to repeatedly flush or evict the target cache lines so that the attackers can monitor them, such as the *Flush* phase in *Flush+Reload* attack and the *Prime* phase in *Prime+Probe* attack. In order to distinguish between loops and

other structures with backward edges, we highlight the loop edges by marking them as type 0.

2. type 1: The tree structure indicates a branch like *if-else* or *switch*. They are frequently used in attack, and is marked as type 1 in our . For example, *Flush+Reload with Spectre* needs to decide whether to train the branch predictor or perform the speculative execution attack according to the current CPU state.

3. type 2: The tree structure in a circle represents a branch in the loop. Some attack steps such as *Reload* phase in *Flush+Reload* or *Probe* phase in *Prime+Probe* need to repeatedly access the target cache lines again and determine whether the victim executes by the cache access time. Therefore, we mark this combination as a special class in our system.

4. type 3: The remaining structures belong to type 3 by default.

According to the rules, we identify the structures in CFG automatically. Each edge *(i,j)* in the HPC-embedded CFG will be labeled as the corresponding type in C. More specifically, type information of edge (i,j) is stored in $C[i,j]$ directly.

With the locality guided program representation generation method, we can highlight the attack locality in both the static CFG data and dynamic HPC data. In the next paragraph, the GNN graph classification technology is used to train the program representations, HPC-embedded CFG, of samples, and finally to identify potential attack programs.

3.3 Locality based cache side-channel attack detection

In this subsection, we introduce the workflow of the attack detection with the proposed program representation generation method.

With the proposed program representation generation method, we can obtain the dataset contains program representation for training and classification.

1. Phase 1: Data collection (Algorithm 4 line 3-5). When given the attack samples and benign samples (where FR is the *Flush+Reload* attack, PP represents the *Prime+Probe* attack, and BN indicates the benign program), necessary static CFG data and dynamic HPC data of each training sample are collected.

2. Phase 2: Locality guided data embedding (Algorithm 4 line 7-15). According to the attack locality, the node embedding and the edge embedding are performed on the collected data to form the HPC-embedded CFG G

3. Phase 3: Dataset generation (Algorithm 4 line 17-18). The transformed HPC-embedded CFG of all samples and their labels are added into the dataset.

4. Phase 4: GNN training (Algorithm 4 line 21). The Enhanced Graph Attention Network (EGAT) [26] is used in the paper for training an attack classifier AG. EGAT is one of GNN schemes, which allows the graph to aggregate the information of neighbor nodes and edges according to the weight, and then to update the feature representation of the current node and edge. We use EGAT in the training since it pays attention to both the edges and the neighbour nodes in the graph. Thus, it can handle the features of HPC data and CFG strcuture encoded into the CFG.

Algorithm 4 Training the GNN classifier for cache side-channel attack detection

Input: The attack samples and benign samples *S(FR,PP,BN)*;
Output: Attack Classifier AC;

```
DATASET=[]
for Each sample S_i in S do
    /*Data Collection*/
    Get CFG of S_i as (Vertices(InstructionAddress, Instruction), Edges)
    Get Extended_HPC_Data as (Timestamp, InstructionAddress, HPCEvent)

    /*Locality guided data embedding as the HPC-embedded CFG G(V,E,A,C) */
    G[V]=CFG[Vertices];
    G[E]=CFG[Edges];
    for Each node V_j in G[V] do
        G[A][j]=SUM( HPC_Data mapped to V_j according to the InstructionAddress )
    end for
    for Each edge E(j,k) in G[E] do
        G[C][j,k]=TYPEOF(The structure to which E(j,k) belongs)
    end for

    /*Append to the dataset */
    DATASET.append(G, label) //The labels corresponding to samples FR, PP, and BN are
    0, 1, 2
end for

AC=EGAT(DATASET)
return AC
```

With the trained GNN graph classifierAC, we can deploy the locality based cache-side channel attack detection solution to the target program. When a new program is launched, our attack detection solution is triggered to collect the target program's CFG and the HPC data, then classify it into benign programs or attach programs like *Flush+Reload* attack, and *Prime+Probe*.

4 Experiments

In this section, we evaluate the locality-based cache side-channel attack detection method proposed in this paper. The experiment is designed to evaluate the accuracy, overhead and so on of our approach.

4.1 Implementation

As shown in the previous sections, in our method, static CFG and dynamic HPC data of target programs are needed for the detection. In our implementation, we use Angr [27], Intel Processor Trace (Intel PT) [28] and Perf [29] to collect such data.

Angr is a python framework for analyzing binaries. It provides a program interface for users to automatically extract the control flow graph of the target program.

As for the dynamic information, both the Intel Processor Trace (Intel PT) [28] and Perf [29] is introduced in our HPC data collection. Intel PT is an extension of the Intel Processor that collects information about software execution. And, Perf is a performance counter profiling tool widely used by existing HPC collection methods. It provides the capability of collecting the HPC data for a target program. It is noted that Intel PT provides an interface provide an interface to work with perf. When Intel PT works, the current executed instruction and address can be recorded, and then perf is also triggered to collect current HPC data [30].

In this way, the necessary static CFG and dynamic HPC data of target programs are collected and can be transformed into the program representation. Then the obtained dataset with program representation of all samples, can be trained by an GNN graph classifier to identify the cache attacks.

4.2 Experiment setup

For the training of the GNN-based classifier for attack detection, we prepare a set of programs as shown in Table 2.

First, we choose two classical cache side-channel attack programs *Flush+Reload* and *Prime+Probe* as positive samples. In detail, we collect 3 *Flush+Reload* implementations and 4 *Prime+Probe* implementations by different authors from Github [31]. To increase the number and diversity of positive samples, we also use mutation technology [32] on these two sets of attack programs. The mutation technology will randomly change the source code with given constraints. It is noted that minor changes such as *replacing logical operators*, *swapping the increment and decrement operators* do not change the CFG of a program, but major changes, such as *Deletes a whole line* may lead to changes of the CFG. After mutation, we obtain 400 *Flush+Reload* samples and 400 *Prime+Probe* samples.

We also collect 400 benign samples from the following sources:

1. SPEC2006: SPEC2006 is a standard performance benchmark. We choose 12 test cases with different degrees of memory access workload from SPEC2006 [33].

2. Leetcode: Leetcode [34] is an online coding platform for professionals. We collect 288 solutions from the platform.

3. "Benign" *Flush+Reload* programs by randomly removing the attack steps: To evaluate the detection effect of our method on highly confused samples, we randomly remove the attack steps from the source code of *Flush+Reload* attack, and generate 100 different samples. The CFGs of obtained samples are highly similar to the original attack program. However, these samples are benign programs as their attack process are incomplete.

Our experiments were conducted on a 3.4GHz*8 core i7 PC with 16 GB of RAM under the Ubuntu 20.04.

4.3 Detection accuracy

We use the 10 fold cross-validation scheme to evaluate our multi-class classification method. The experimental results are reported from following aspects: *Accuracy*, *Recall*, *F1-Score* as calculated below:

$$Accuracy = \frac{TP}{TP + FP} \tag{1}$$

Table 2: The dataset situation required for different attack detection methods

Type	Cases	Number
The positive samples	Flush+Reload	400
	Prime+Probe	400
The negative samples	Begin Programs (SPEC2006)	12
	Begin Programs (Leetcode)	288
	Begin Programs ("Benign" *Flush+Reload*)	100

Table 3: The results of the GAT classification

Approaches	Accuracy	Recall	F1-Score
Our Method	99.44%	99.50%	99.47%
Congmiao et al [15] (LR-based)	84.74%	74.41%	75.43%
Maria et al [16, 17] (SVM-based)	96.26%	96.28%	96.23%
Picek et al [35] (CNN-based)	86.45%	86.67%	86.54%

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F1 - Score = \frac{2 \times Accuracy \times Recall}{Accuracy + Recall} \tag{3}$$

,where TP is the number of attack programs that correctly identified as attacks, FP represents the number of benign programs that were misclassified, and FN indicates the number of attack programs that were misclassified.

Meanwhile, we also measure the *Accuracy*, *Recall*, and *F1-Score* of different typical cache side-channel attack detection methods [15, 16, 17]. The results can be found in Table 3. The experimental results show that our scheme is able to achieve 99.44% accuracy, and 99.47% F1-Score. It can be 3%∼15% better than other approaches with fewer false alarms (higher *Recall* and higher *F1-Score*).

4.4 Performance overhead

Our method needs to collect the runtime HPC data and extract the CFG of the target program. Therefore, we also examine the performance overhead of the monitored target programs and

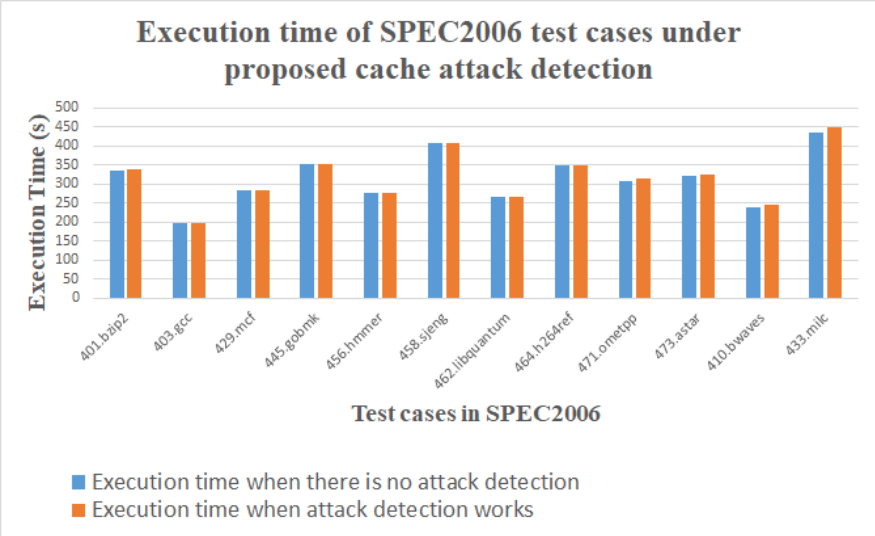

Figure 8: Execution time of SPEC2006 test cases with & without our detection method running

the entire system.

First, we run the target program with and without our detection method, and to record the time cost. The average execution time of the target programs without our detection method is 4764.417s. If our detection method is deployed, the average execution time of these target programs is 5383.32s. The performance overhead of the target programs is increased by 12.98%.

On the other hand, we also evaluate how our detection method affects the running system. SPEC2006, which is used in the benchmark of our experiments, is a standard performance benchmark widely used for evaluating the computer's performance. Therefore, we use SPEC2006 to evaluate the system performance with the effect of our attack detection.

In this experiment, we run programs in SPEC2006 in two different scenarios: scenario 1, a clean system that our detection method is not running, scenario 2, a system that our detection method is checking another program at the same time.

We record the time cost of the SPEC2006 program under these two scenarios correspondingly. The data is shown in Figure 8. In this figure, the blue rectangles show the execution time of programs in SPEC2006 in scenario 1, and the red rectangles show the time cost of programs in SPEC2006 in scenario 2. We can see, the height of these two sets of bars are very close to each other. Actually, after calculation, the average performance cost is only 0.98%. This confirms our claim that the detection method proposed in this paper has a low impact on performance, and the performance cost of our method is acceptable.

5 Conclusion and Future Work

Based on the observation of the phenomenon of attack locality, we propose a new program representation that integrates the information of dynamic runtime data and the static control flow of a target program. Then, we can use GAT learning of the new program representation to achieve cache side-channel attack detection with high accuracy and low overhead. In the future, we plan to keep investigating the "locality" on the data level, to mine the data relationships

between different basic blocks to make the detection more accurate and scalable.

References

[1] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[2] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[3] Colin Percival. Cache missing for fun and profit, 2005.

[4] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[5] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[6] Zecheng He and Ruby B Lee. How secure is your cache against side-channel attacks? In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 341–353, 2017.

[7] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[8] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[9] Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215. IEEE, 2014.

[10] Lars Müller. Kpti a mitigation method against meltdown. *Advanced Microkernel Operating Systems*, page 41, 2018.

[11] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[12] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[13] Sam Ainsworth and Timothy M Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 132–144. IEEE, 2020.

[14] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2018.

[15] Congmiao Li and Jean-Luc Gaudiot. Detecting malicious attacks exploiting hardware vulnerabilities using performance counters. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 588–597. IEEE, 2019.

[16] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Rao Naveed Bin Rais, Vianney Lapotre, and Guy Gogniat. Run-time detection of prime+ probe side-channel attack on aes en-

cryption algorithm. In *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–5. IEEE, 2018.

[17] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Muneeb Yousaf, Umer Farooq, Vianney Lapotre, and Guy Gogniat. Machine learning for security: The case of side-channel attack detection at run-time. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 485–488. IEEE, 2018.

[18] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[19] Jonas Depoix and Philipp Altmeyer. Detecting spectre attacks by identifying cache side-channel attacks using machine learning. *Advanced Microkernel Operating Systems*, 75, 2018.

[20] Congmiao Li and Jean-Luc Gaudiot. Online detection of spectre attacks using microarchitectural traces from performance counters. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 25–28. IEEE, 2018.

[21] Zhuo Ma, Haoran Ge, Yang Liu, Meng Zhao, and Jianfeng Ma. A combination method for android malware detection based on control flow graphs and machine learning algorithms. *IEEE access*, 7:21235–21245, 2019.

[22] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 52–63. IEEE, 2019.

[23] Intel. Perf events list. https://oprofile.sourceforge.io/docs/intel-corei7-events.php, 2021-8-22.

[24] Zirak Allaf, Mo Adda, and Alexander Gegov. A comparison study on flush+ reload and prime+ probe attacks on aes using machine learning approaches. In *UK Workshop on Computational Intelligence*, pages 203–213. Springer, 2017.

[25] Zhongkai Tong, Ziyuan Zhu, Zhanpeng Wang, Limin Wang, Yusha Zhang, and Yuxin Liu. Cache side-channel attacks detection based on machine learning. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 919–926. IEEE, 2020.

[26] Liyu Gong and Qiang Cheng. Exploiting edge features for graph neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9211–9219, 2019.

[27] Angr Team. Angr binary analysis tool. http://angr.io/, 2021-7-22.

[28] Intel Developer. Intel processor trace. https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html, 2021-7-20.

[29] Linux Kernel. Linux perf tools. https://perf.wiki.kernel.org/index.php/, 2021-7-20.

[30] Linux Man Document. Intel processor trace within perf tools. https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html, 2021-7-20.

[31] Github Team. Github. https://github.com/, 2021-7-22.

[32] Niels Lohmann. Mutate_cpp. https://github.com/nlohmann/mutate_cpp, 2021-7-22.

[33] Karthik Ganesan, Jungho Jo, and Lizy K John. Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 33–44. IEEE, 2010.

[34] LeetCode Team. Leetcode oj platform. https://leetcode.com/explore/, 2021-9-7.

[35] Stjepan Picek, Ioannis Petros Samiotis, Jaehun Kim, Annelie Heuser, Shivam Bhasin, and Axel Legay. On the performance of convolutional neural networks for side-channel analysis. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 157–176. Springer, 2018.